Gene-Environment Interaction Analysis Using Graphic Cards

Daniel Berglund

2 February 2015

Abstract

Genome-wide association studies(GWAS) are used to find associations between genetic markers and diseases. One part of GWAS is to study interaction between markers which can play an important role in the risk for the disease. The search for interactions can be computationally intensive. The aim of this thesis was to improve the performance of software used for gene-environment interaction by using parallel programming techniques on graphical processors. A study of the new programs performance, speedup and efficiency was made using multiple simulated datasets. The program shows significantly better performance compared with the older program.

Sammanfattning

Genome-wide association studies(GWAS) används för att hita associationer mellan genetiska markörer och sjukdommar. En del av GWAS är att studera interkation mellan markörer vilket kan spela en viktig roll för sjukdomsrisken. Målet med det här arbetet var att förbättra prestanda av mjukvaran som används för gen-miljö interaktion genom att använda tekniker för parrallel programmering på grafikkort. En studie av det nya programmets prestanda, uppsnabbning och effektivitet genomfördes på flera simulerade datasätt. Programmet har signifikant bättre prestanda jämfört med det äldre programmet.

Acknowledgements

I would like to express to my thanks to Lilit Axner and Henrik Källberg for providing and supervising this project. I would like to thank KIRC for all the interesting discussions and welcoming atmosphere, it has been an interesting time from which I have learned a lot. I would like to thank Michael Schliephake and PDC for the support and allowing me to use the Zorn cluster.

Contents

Chapter 1

Introduction

1.1 Outline

The first chapter of this thesis contains a short introduction to the area and the terminology. The second chapter gives the theoretical background. It starts with the statistics and algorithms and then describes the computer architecture and software design. The third chapter explains the current algorithm and also explains the design of the program that was developed in this thesis called CuEira. Chapter four provides the results and comparisons of the program performance. The fifth chapter contains the discussion of these results. Finally the last chapter discusses future work.

1.2 Genome-wide association studies

One type of study to find associations between genetic markers and diseases or other traits in different individuals is genome-wide association study(GWAS). Most GWAS do not study interactions between the genetic markers or between the genetic markers and environmental factors [1, 2]. Investigating gene-gene interactions recently has become more common [1], however gene-environment interactions are still uncommon in studies [2]. Interactions between genes and environmental factors are considered to be important for complex diseases such as cancer or autoimmune diseases [1, 2, 3, 4]. In general a complex disease develops due to a combination of factors, not just a single gene or environmental factor [5]. Examples of environmental factors are smoking, physical activity and so on.
GWAS usually has a study design that is either cohort or case-control [5]. In cohort study a sample of a population is followed and over time some individuals will develop the disease of interest [6]. In case-control studies two groups are compared with each other to find the risk factors [6]. One of these two groups consists of individuals with the disease and the other group consists of individuals similar to the first group but who do not have the disease [6].

A typical GWAS consists of tens of thousands of individuals and up to several millions of genetic markers [1, 7]. In gene-gene interaction GWAS few studies investigate more than the second order interaction because of the high number

of possible combinations. There are some algorithms that can investigate higher order interactions however these have drawbacks [8, 9, 10].

1.3 Genetics

The genetic information is stored in *chromosomes* each consisting of a *DNA molecule* [11]. Each chromosome comes in a pair where both chromosomes are nearly identical, except for the chromosomes related to gender [11]. The DNA molecule is a double stranded helix of four nucleotide bases, *adenine*(A), *cytosine*(C), *guanine*(G) and *thymine*(T). They are always paired as A-T and G-C. *Genes* are sections of these DNA molecules. The pair of genes in the chromosomes is a *genotype*. The observable product of the genotype is called *phenotype*, e.g. eye colour, fur patters.

A position of the DNA or a gene is a *locus* [11]. A variation of the same gene or locus is an *allele* [11]. In classic(i.e. Mendelian) genetics the effect of an allele on a phenotype can be either *dominant*, *recessive* or *co-dominant*. Dominant means that if the allele is present the phenotype is expressed. For recessive the allele needs to be present in both chromosomes to be expressed [11]. Co-dominant is when both alleles are expressed, when different alleles are present this usually produce some kind of intermediate state [11]. An example of co-dominance is the human blood groups [11].

Genotype	Dominant	Recessive
AA	Effect	Effect
GA	Effect	No Effect
GG	No effect	No effect

Table 1.1: *The phenotype based on the genetic model and if the allele with the effect is A*

1.3.1 Major, Minor and Risk Allele

The the allele that is least common using all individuals(both cases and controls) is the *minor allele*, the one that is most common is the *major allele*. The frequency of the minor allele is the *minor allele frequency*(MAF) [12, 3]. If the MAF is too low an analysis often will be of little value so all SNPs with MAF below a threshold are usually skipped. 5% is common value of the threshold [7, 3].

To determine if the genetic risk is present for each individual one of the alleles needs to be chosen as the *risk allele*. GEIRA and JEIRA defines the risk allele as the minor allele in cases if the MAF of cases is greater than the MAF of controls, otherwise the major allele in cases is used [13, 12]. However JEIRA and GEISA does not calculate the risk allele according to that definition, see appendix C for the details. In this thesis the risk allele will be defined as the allele that has higher frequency in cases than controls. It is similar to the definition using MAF in most cases. Appendix C has more detailed comparisons.

The presence of genetic risk based on the risk allele and the genetic model can be coded as a variable than then be can used in a statistical model. For dominant or recessive genetic model this means that the variable is binary since the risk is either present or not [3]. See table 1.1 for how the coding is based on the model. For co-dominant model the number of risk alleles in the individual is used as the variable, 0, 1 or 2 [3].

1.4 Interaction

There are several ways to define interaction. The overall goal is often to detect if *biological* interactions are present. *Biological* interaction is when different factors co-operate through a physiological or biological mechanism and cause the effect, e.g. the disease. The information about the factors can then be used to explain the mechanisms involved in causing the disease and possibly help to find cures for them. However biological interaction is not well defined and thus it is not possible to calculate it directly from data. [14, 5, 15]

That is why statistical interaction is used and it is assumed that the presence of statistical interaction implies the presence of biological interaction
Statistical interaction on the other hand is well defined. However it is scale dependent, i.e. interactions can appear and disappear based on transformations of the given data. Statistical interaction also depends on the model used. The common way to define statistical interaction is to consider the presence of a product term between the variables in the model, this is referred to as *multiplicative* interaction. For instance for a linear model

$$f(x, y) = ax + by + cxy + d \tag{1.1}$$

c is the product term that represents multiplicative interaction between variables x and y so the test for multiplicative interaction is to test if $c = 0$ [3, 14, 5]. In some cases it is instead tested as $a = b = c = 0$, this is testing for *association allowing for interaction* [1].

Additive interaction is another kind of statistical interaction that is broader than multiplicative interaction. It is when the total effect of the interacting factors present is larger than the sum of their separate effects. A more precise definition can be found in section 2.1.3. It implies biological interaction as defined by Rothman [5], which is sometimes called *causal interdependence* or *causal interaction* [16]. Additive interaction can find more possible interactions than multiplicative and since it is closer to biological interaction it is often chosen in epidemiological studies [14].

1.5 GEIRA, JEIRA and GEISA, the Aim of the Project

Various tools can be used to for searching for interaction in genetic data. *GEIRA* [3] is one such tool which uses logistic regression, additive interaction and bootstrap to asses the quality of the statistical models [3]. It comes in two versions, one in R and one in SAS. To improve speedup it was later implemented in Java as *JEIRA* [12]. JEIRA is also parallel which made it significantly faster [12]. *GEISA* [13] is another tool based on JEIRA however it uses permutation testing instead of bootstrap to asses the certainty of the models [13]. The structure is explained in chapter 3. Bootstrap calculates the certainty by resampling the data [17]. Permutation tests works in a similar way but only randomizes the outcomes [18].

The aim of this thesis is to make the gene-environment interaction analysis faster by using graphics processors and to be able to handle larger amounts of data. The program is written in C++ and CUDA.

Chapter 2

Background

This chapter will explain some of the background starting with the statistical background, continuing with computer architecture and software design. Finally is a section about the algorithms.

2.1 Statistical Background

This section will explain some of the statistical background with focus on categorical data.

2.1.1 Contingency Tables

A contingency table is a matrix used to describe categorical data [17]. Each cell contains a count of occurrences for a specific combination of variables [17]. Table 2.1 is an example of an 2 x 2 table. From this table we can for instance see that 688 smokers got lung cancer. Contingency tables are the basis for various statistical tests to model the data [17].

	Lung cancer	No lung cancer
Smoker	688	650
Non smoker	21	59

Table 2.1: *Contingency table describing the outcome of a study, from [17], page 42*

2.1.2 Relative Risk and Odds Ratio

From a contingency table it is possible to calculate some useful measures such as the risk of getting the outcome based on exposure [17]. The risk for a row i in the table will be referred to as π_i. For table 2.1 the risks π_1 and π_2 is

$$\pi_1 = \frac{688}{688 + 650} = 0.51 \tag{2.1}$$

$$\pi_2 = \frac{21}{21 + 59} = 0.36 \tag{2.2}$$

To compare different risks, for instance between smokers and non smokers, the ratio of the risks is used [17]. It is called *relative* risk(RR) is defined as [17]

$$RR = \frac{\pi_1}{\pi_2} \tag{2.3}$$

So for the table 2.1 the relative risk of getting lung cancer based on exposure to smoking is

$$RR = \frac{0.52}{0.36} = 1.96 \tag{2.4}$$

This means that the risk of getting lung cancer for a smoker is almost twice as high for a non smoker in this data. Another useful measure is *odds* and *odds ratio*(OR) [17]. The odds is

$$\Omega = \frac{\pi}{1-\pi} \tag{2.5}$$

The odds are non-negative and $\Omega > 1$ when the outcome is more likely than not [17]. So for $\pi = 0.75$ the odds is $\Omega = \frac{0.75}{1-0.75} = 3$. This means that the outcome is 3 times more likely to occur than not. ORs can be used as an approximation of RR in case control studies because RR can not be estimated in that type of studies [19]. Cohort studies on the other hand can give estimates of RR [19]. OR is the ratio of the odds just as RR is the ratio of the risks [17].

$$\theta = \frac{\Omega_1}{\Omega_2} \tag{2.6}$$

In the case when the outcome is a disease or similar variables with odds ratio below one are called *protective* and when it is above one it is a *risk factor* [20].

2.1.3 Additive Interaction

The precise definition of additive interaction is the divergence from additive effects on a logarithmic scale, e.g [5].

$$OR_{both\ factors\ present} > OR_{first\ factor\ present} + OR_{second\ factor\ present} - 1 \quad (2.7)$$

A third variable is used to model the interaction for both multiplicative and additive interaction. Multiplicative interaction as explained in section 1.4 does not include the effects from the factors, i.e. the main effects, in the interaction variable. However for additive interaction the interaction variable models the whole effect including the main effects. This means that the third variable for multiplicative and additive interaction are identical, however for additive interaction the first and second variable is zero when both factors are present [12]. The coding for additive interaction is summarized in table 2.2 [12].

Factor present	First variable	Second variable	Interaction
None	0	0	0
First	1	0	0
Second	0	1	0
Both	0	0	1

Table 2.2: *Coding of the variables for LR*

2.1.4 Statistic Measures for Additive Interaction

Based on the statistical model and its corresponding ORs there are some measures that can be calculated. These measures show various properties of the additive interaction [3, 20]. They are defined using RR however as mentioned before in section 2.1.2 OR can be used to approximate RR.

Relative excess risk due to interaction(RERI) is how much of the risk is due to interaction [5, 20]. It is defined as [5, 20]

$$RERI = RR_{11} - RR_{10} - RR_{01} + 1 \quad (2.8)$$

Attributable proportion due to interaction(AP) is similar to RERI however is the proportion relative to the interaction relative risk [5, 20].

$$AP = \frac{RERI}{RR_{11}} = \frac{1}{RR_{11}} - \frac{RR_{10}}{RR_{11}} - \frac{RR_{01}}{RR_{11}} + 1 \quad (2.9)$$

Synergy index(SI) is the ratio of the combined effects and the individual effects [5, 20].

$$\frac{RR_{11} - 1}{RR_{10} + RR_{01} - 2} \quad (2.10)$$

By using the definition of additive interaction it can be shown that additive interaction is present when $RERI > 0$, $AP > 0$ and $SI > 0$ [20].

2.1.4.1 Recoding of Protective Effects

The measures for additive interaction RERI, AP and SI explained in the previous section are developed for risk factors, i.e. $OR > 1$, which causes problems if a factor is protective [20]. This can be solved by *recoding*, it switches the group of individuals without both factors, i.e. the *reference group*, with the group that has the lowest OR [20]. There are three possible combinations of the factors excluding the reference group shown in table 2.3. Switching a group changes the level of the factors for all in individuals in the group to the reference groups level, and switches the individuals in the reference group with the groups level.

Genetic factor	Environment factor	Recoding case
0	0	N/A
1	0	1
0	1	2
1	1	3

Table 2.3: *Cases of recoding*

2.1.5 Confounders and Covariates

Confounding is one of the central issues in design of epidemiological studies [15, 5]. It is when the effect of the exposure is mixed with the effect of another variable [15, 5, 17]. So if we do not measure the second variable, the effect of the first would be estimated as stronger than it really is [15, 5]. The second variable is then a *confounder* [15, 5]. Several methods in epidemiology are about avoiding or adjusting for confounding [15, 5]. Sometimes these variables needs to be incorporated into the models. *Covariates* are possible confounders or other variables that one wants to adjust for in the model. Sometimes covariates are called *control variables* [15, 5].

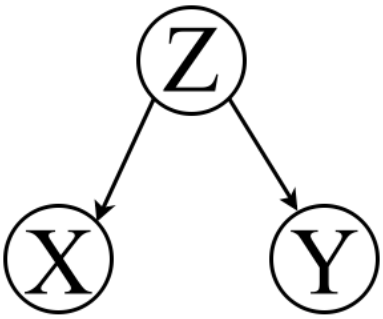


Figure 2.1: *Illustration of a simple case of confounding. If we do not observe Z we might falsely find an association between X and Y*

For instance if a study would look into the effects of yellow teeth on lung cancer they would likely find an effect. However it would be due to the confounder smoking since smoking causes yellow teeth and lung cancer [21].

2.1.6 Multiple Testing

It is not uncommon to test many hypothesis on the same data and in the case of GWAS it can be millions, or in the case of gene-gene interaction even trillions, of tests [1]. If one continues to test one should eventually find something that is significant due to random chance [22]. With the common significance threshold of 5% it is expected to get 1 in 20 false positives under the assumption that the null hypothesis is true. The problem arises from the fact that the hypothesis tests are dependent on each other since they use parts of the same data [22]. This is the multiple testing problem and if it is not corrected for many false positives might be found [22].

Bonferroni correction is one of the simplest method and viewed as a conservative way to correct for this problem [22]. It simply divides the significance threshold by the number of hypothesis tested [22]. However the number of hypothesis made is not always clear. With a two-stage analysis, is the number of hypothesis the number of tests done in both stages combined, the number made in the first stage or the second stage.

2.2 Computer Architecture Background

This section explains some of the computer architectures used. A large part of this chapter is focused on the optimization techniques that are in use. For the purpose of this thesis it is best to think of the computer consisting of four main parts, the processor, the data storage, the memory and the accelerator.

2.2.1 Central Processing Unit

Central processing unit(CPU) is the part of the computer that executes instructions one by one [23]. Most modern CPUs have multi-core architecture meaning they consist of several processors [23]. Each core can perform tasks independent of each other. Multi-core architecture affects the way the programs are made because they need to be parallel to get maximum speed. More about this in section 2.2.2.

The figures 2.2 and 2.5 shows how the CPU is divided into areas and that most of it is not used for calculations. A large part of the area is used for various optimizations.

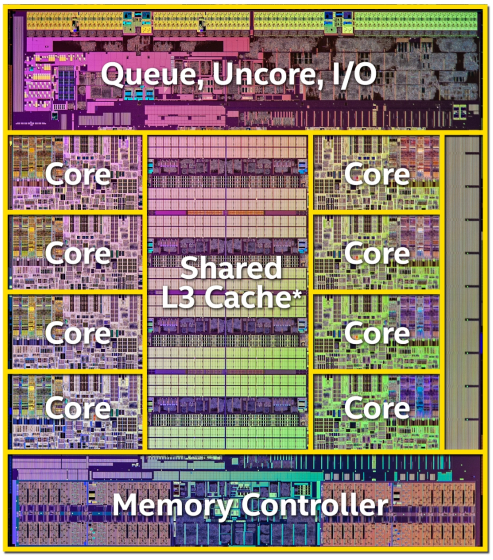


Figure 2.2: *The layout of the Haswell architecture i7-5960x, from [24]*

2.2.2 Concurrency and Threads

Concurrency means doing multiple things and the same time, instead of *sequential* [25, 26]. This can cause various problems such as the same object being accessed at the same time. The dinning philosophers is an concurrency problem that can used to illustrate some of these problems. A group of philosophers is sitting at a round table, each has a plate of spaghetti in front of them and there is a fork between each pair of philosophers [26]. The philosophers alternate between thinking and eating. However they can only eat if they have both the right and left fork [26].

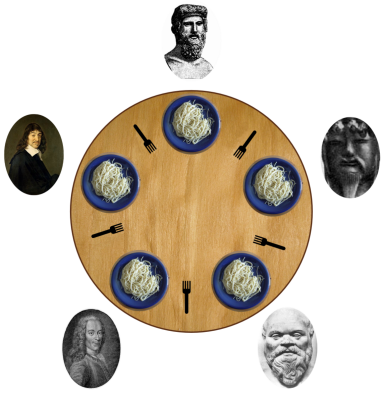


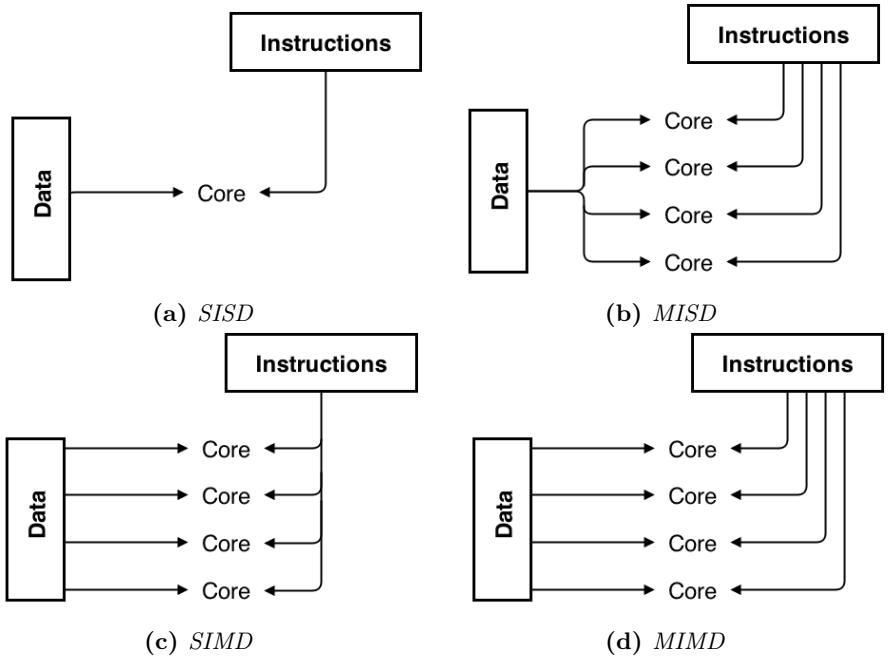
Figure 2.3: *Illustration of the dining philosophers problem. Wikipedia Commons*

A potential proposal [26] for behaviour instructions could be:

- Think
- Wait for a fork to become available and pick up that fork
- Wait for and pick up the other fork
- Eat
- Put down the forks one by one
- Go back to thinking

The problem is that this set of instructions can lead to a state where everyone is holding one fork and waiting to get the other one [26]. But no one is done eating so no forks will become available. The system is then locked into its state, this is called a *deadlock* [26, 23]. A potential solution to the problem in this case is to introduce a person that dictates if a philosopher is allowed to eat or not [26].

There are also other potential problems that can arise when using concurrency. A *race condition* occurs when the result depends on what affects it first, it is a race between them on who can reach it first [23]. *Locks* can be used to prevent situations as the ones described above by making sure only one thread at a time can access the object [23].



(a) *SISD* **(b)** *MISD*

(c) *SIMD* **(d)** *MIMD*

Figure 2.4: *Flynns taxonomy of parallel architectures [23]*

In computer science concurrency occurs when instructions are separated in different *threads*. The architectures can be divided in a taxonomy with four categories including the sequential. They are illustrated in figure 2.4.

- *SISD*(Single-Instruction,Single-Data) is the sequential and is as the name says, single instructions on single data [23].
- *SIMD*(Single-Instruction, Multiple-Data) is applying the same instruction on different data [23]. SIMD is also called vectorisation [23].
- *MIMD*(Multiple-Instruction, Multiple-Data) applies different instructions on different data [23].
- *MISD*(Multiple-Instruction, Single-Data) performs different operations at the same piece of data, this paradigm is uncommon [23, 27].

2.2.3 Memory, Caching and Optimizations

Various optimizations have been introduce to the CPUs over the years [28, 23]. Some are common and almost all CPUs have them, others are unique to a specific vendor or CPU [23]. Figure 2.5 show the layout of a CPU core. This section explains some of the more common optimizations.

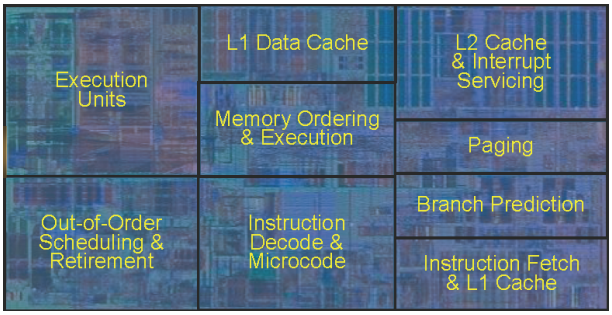


Figure 2.5: *Layout of a CPU core, Intel Nehalem i7, from [29]*

In general retrieving data from memory is relatively slow compared to how fast the processor works [23, 28]. To solve this problem a *cache* was introduced [28]. It stores recently used data in a very small and very fast memory that resides on the CPU chip itself [28]. The caches varies in sizes, the latest ones are around 16-20 mB, for instance Intels i7-5960x with 20 mB [30]. The cached data can then be reused without waiting for the main memory to fetch it. However when the data is not in the cache it costs time to fetch it from the main memory, that is a *cache miss* [28]. Avoiding cache misses is important for the speedup [28]. Modern CPUs commonly have more than one cache, most have three [23]. They are named L1 to L3, with L1 being the one fastest and smallest and L3 the largest and slowest [23].

Modern multi-core processors have several L1 caches, one per core [23]. Commonly each core has a L2 cache as well, however it can be shared in some architectures [23]. The L3 cache is almost always shared between all cores [23]. The multiple caches used causes a problem called *cache coherency*. It is when the data modified in one cache needs to be updated in all the caches in order for them to use the new value [23].

One of the memory optimization methods somewhat related to caches is *prefetching* [23, 28]. Instead of fetching just the data requested by the current operations it also fetches the surrounding data [28]. Even if the data is not needed at the given point in time, the chance that the surrounding data will be used soon is high since it is common to iterate over arrays and similar structures [23].

When a program iterates over an array the first value is a cache miss since it is not in the cache. The following values are prefected and cached as well. However if the array is too long to be prefetched and cached completely and the CPU hits a position not in the cache then a new cache miss happens [28]. Usually matrices are stored sequential in memory [28]. For instance a two dimensional matrix is stored as a one dimensional array. Different programming

languages store them differently, either column by column or row by row [28]. Row by row is called *row major* and column by column is *column major.*

Another optimization is *branch prediction* and *speculative execution.* When the operation instructions cause divergence, e.g. *if* statements, the CPU has to wait for the previous instructions to finish to see which path it should diverge to [28]. This can mean a significant loss of time, however because of *branch prediction* and *speculative execution* the loss of time can be prevented in some cases [28]. A part of the CPU stores the outcomes from these divergences and when the same divergence is encountered again it uses the stored outcomes to make a guess of what do to next [28]. The CPU then loads the instructions and speculatively executes the instructions, i.e it executes instructions that might not actually be needed [28]. If the guess was correct the results are kept, on the other hand if it was wrong the CPU starts again from the correct path and discards the incorrect results [28].

2.2.3.1 Cache-Friendly Code

Correct and incorrect usage of caches can effect the performance significantly. In section 6.2.1 in What Every Programmer Should Know About Memory [28] an example of how much speed that can be gained by optimizing matrix matrix multiplication. The variables here have been renamed for clarity. The simple version of the algorithm is shown in algorithm 1 and the version with improved cache use is shown in 2.

By transposing the *matrix2* matrix it becomes more cache friendly by using prefectching of surrounding data the way that *matrix1* already is [28]. In the simple version, algorithm 1 *matrix1* is looped along the way it is stored in the memory while *matrix2* is not. The transpose itself can be eliminated while also making it use the cache better by reading in the correct amount of data, this is the size SM in the code [28]. The optimized version is shown in 2. It took 17.3% of the original time [28]. On top of that by using some additional features of the CPU such the CPUs SIMD capabilities more speedup is gained increasing it to 9.47% of the original time [28].

Algorithm 1: Basic algorithm for matrix matrix multiplication of two $N \times N$ matrices

Data: Three $N \times N$ matrices, *matrix1*, *matrix2* and *result*
i,j,k are indices spanning from 0 to $N - 1$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j< N; ++j)
    {
        for (k = 0;k < N; ++k)
        {
            result[i][j]+ = matrix1[i][k] * matrix2[k][j];
        }
    }
}
```

Algorithm 2: Matrix matrix multiplication algorithm optimized

Data: Three $N \times N$ matrices, *matrix1*, *matrix2* and *result*
rresult, *rmatrix*1, *rmatrix*2 refers to parts of the matrices, they are used to eliminate common expressions
SM is the number of matrix elements that fits in the cache

```
for (i = 0; i < N; i += SM)
{
    for (j = 0; j < N; j += SM)
    {
        for (k = 0; k < N; k += SM)
        {
            for (i2 = 0, rresult = &result[i][j], rmatrix1 = &matrix1[i][k];
            i2 < SM; ++i2, rresult += N, rmatrix1 += N)
            {
                for (k2 = 0, rmatrix2 = &matrix2[k][j]; k2 < SM; ++k2,
                rmatrix2 += N)
                {
                    for (j2 = 0; j2 < SM; ++j2)
                    {
                        rresult[j2]+ = rmatrix1[k2] * rmatrix2[j2];
                    }
                }
            }
        }
    }
}
```

2.2.3.2 Independent Instructions and Aliasing

Sometimes there can be several independent instructions, e.g. when adding two vectors together, this allows optimizations for *instruction level parallelism* [23]. The order of the instructions can be ignored which for instance means that if an instruction needs to wait for some data another instruction that has its data available could be executed while waiting [23]. It is an optimization called *out of order execution* [23]. There are also other possible optimizations [23].

In example 1 the instructions are independent because they use different variables so they can be executed in any order. If the variables b and c needs to be fetched but e and f are available due to previous instructions then the CPU could execute the second addition first by using out of order execution. However if an operation later depends on the variable a or d then that instruction would have to wait until the additions have been executed.

Example 1: Independent Instructions

Data: Six int variables, $a, .., f$

$a = b + c$
$d = e + f$

Pointer aliasing can prevent the compiler from making certain optimizations, e.g. out of order execution mentioned above. Pointer aliasing is when the same memory area can be accessed using different variables [23]. This becomes a problem if an set of instructions use these variables in the same area of the program. The relative order of operations between these operations then has to be preserved. The compiler does have the information if aliasing occurs or not which means the compiler has to treat all situations where it can occur as if it does [23].

If the integers in example 1 are changed to pointers then aliasing can occur because some of these pointers could point to the same memory. For instance if a and e points to the same memory then the result would be wrong if the second line is executed before the first. This would force the instructions to be executed in order or the result would likely be incorrect.

Example 2: Pointer Aliasing

Data: Six int pointers, $a, .., f$

$*a = *b + *c$
$*d = *e + *f$

Some programming languages, e.g. Fortran, disallow some types of aliasing while others, e.g. C/C++, allows aliasing [23]. The compilers for the languages in the second case commonly have an option to disable aliasing. However for languages that don't allow aliasing, or if it is disabled, it is then the programmers

responsibility to make sure that aliasing does not occur or the results can be wrong [23].

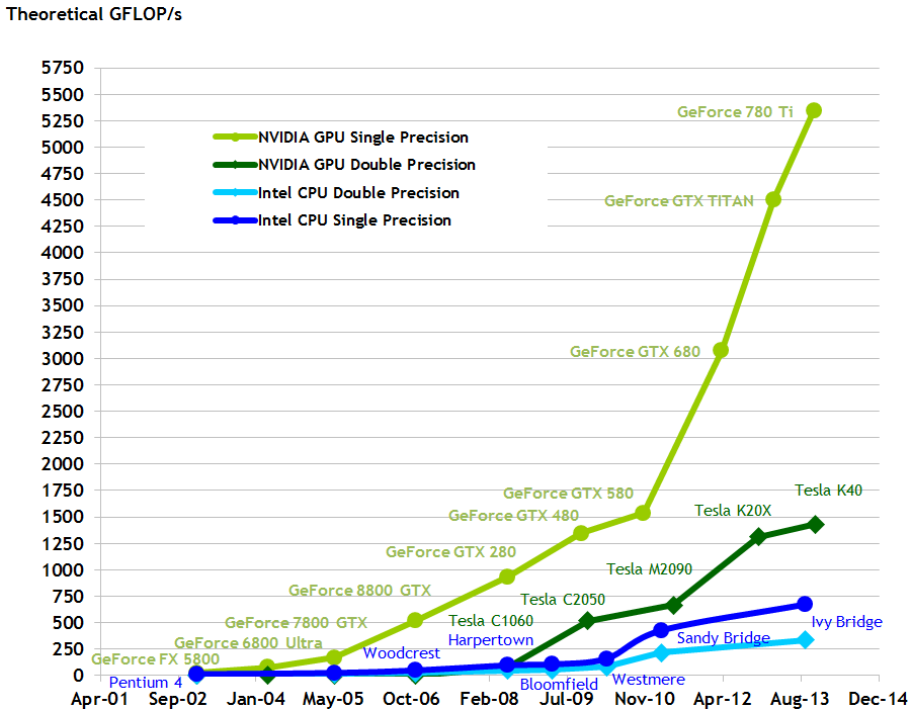


Figure 2.6: *Theoretical throughput of some NVIDIA GPUs and Intel Processors, from [31]. However the chart does not contain a Xeon Phi processor.*

2.2.4 Accelerators, GPU and Xeon Phi

Accelerators are separate parts made to do one task and do it well. They are usually highly specialised and perform badly outside of the tasks they were designed for. The two accelerators explained here is graphical processing unit(GPU) and Intel Xeon Phi [31, 32]. GPUs have mostly been used for games and other graphics related problems however have become more popular for general computing the last 10 years due to their high number of cores [31]. Intel Mic is much more recent with prototypes in 2010 and the first generation released in 2010 [32]. It is Intels response to the GPUs and has elements of both GPU and CPU [32]. This section will focus on explaining some parts of the architecture of the GPUs and Xeon Phi. How to use the GPUs are talked about in more detail in section 2.3.

GPUs are made to process images in a fast manner so that games and similar tasks runs smoothly [31]. Because GPUs are made for image processing they are GPUs have a large number of cores(1000-2000) [31, 33]. Each core is slower than a core in a CPU, however the number of cores makes up for it. The downside is that it's slower than a normal CPU for sequential tasks so the program has to take advantage of the extreme parallelism to get good speed [34]. The memory of the GPU is separate from the computers main memory which means that the data needs to be transferred between the memories which increases the programs complexity [31].

The GPU cores work in a manner that is similar to SIMD [31]. The cores also have fewer optimizations than CPUs, this means that they can then devote a larger portion of the chip to the calculations as seen in figure 2.8. For instance NVIDIAs GPUs lack optimizations such as branch prediction and speculative execution talked about in section 2.2.3 [31]. Their caches are also much smaller than the CPUs [31]. Another inheritance from the image processing is that they are much faster with single precision than with double precision data types [31, 33]. A CPU drops slightly in performance when using double precision however much less than a GPU as seen in figure 2.6. Single and double precision refers to how many bytes are used to store the decimal numbers, fewer bytes means less numerical precision.

Intels answer to the increasing popularity of GPUs for general computing is Xeon Phi [32]. It has elements of GPU architecture however is still more like a CPU. It has separate memory and many cores(50+) [32]. It is MIMD which can be good for codes that diverge a lot, for ease of use however it has the possibly to use more SIMD like structures [32]. It is possible to only use the Xeon Phis memory and avoid the memory transfers that way. The memory is relatively small compared to the main memory so if the program needs a lot of memory transfers are likely needed. The cores of the Xeon Phi lacks some of the normal optimizations, e.g. out of order execution [32].

The choice between different accelerators depends largely on the algorithm, performance requirements, etc. For instance if the algorithm contains many points of divergence Xeon Phi is likely faster because Xeon Phi is MIMD while GPU is SIMT. That Xeon Phi is MIMD also means it can be applied to a broader set of

problems [32]. A one on one comparison of speed might not the correct choice either because of the different prices of the hardware and power consumption. Speed per power consumption or price can be more relevant.

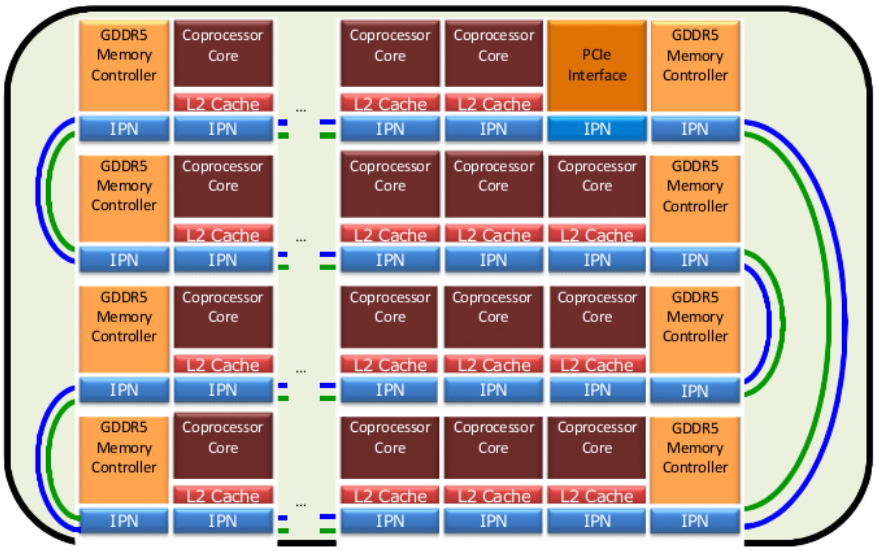


Figure 2.7: *Knights Corner silicon layout, from [35]*

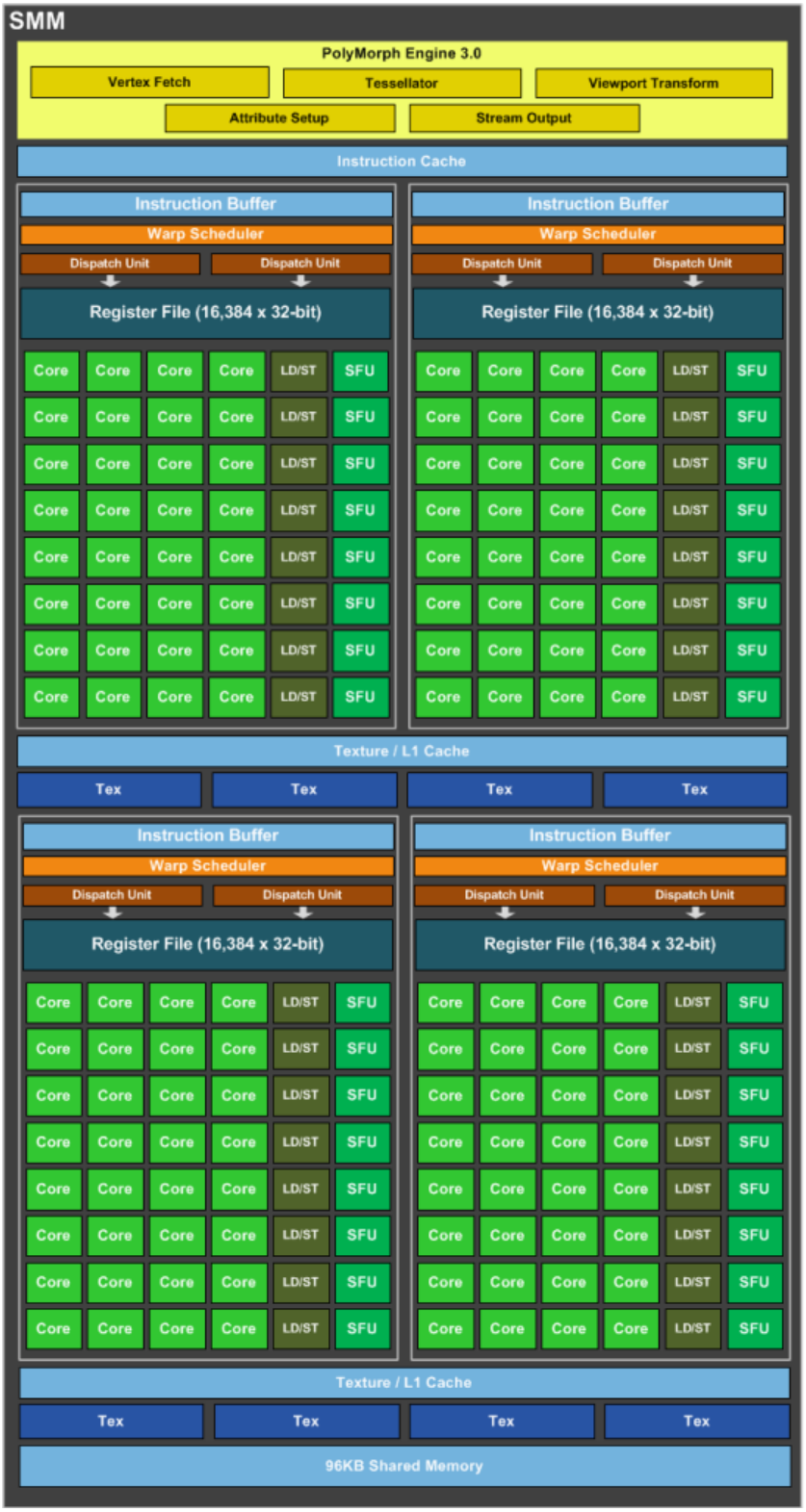


Figure 2.8: *Layout of the Maxwell architecture, from [36]*

2.2.5 Clusters

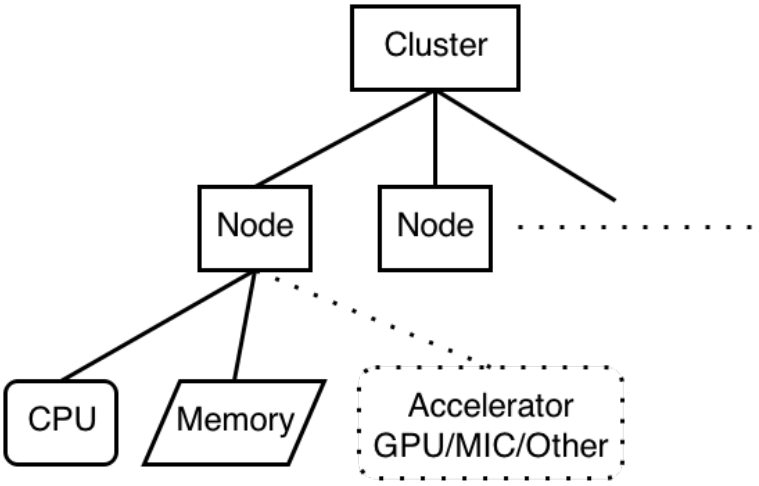


Figure 2.9: *Basic overview of a cluster*

Clusters are collections of computers that can work together in several ways and are so tightly connected that they can usually be viewed as a single system. They communicate over a shared network but have separate memory and processors. Storage is usually shared. A computer in a cluster is called a *node*. [25, 23]

2.3 CUDA programming model

This section is about GPU programming in more detail using NVIDIAs CUDA (Compute Unified Device Architecture) [31]. CUDA extends C and C++ with some additional functionality that can be used to perform operations on NVIDAs GPUs. It provides a separate compiler to compile the GPU code called nvcc [31]. Different GPUs support different CUDA functions, each NVIDIA GPU has a value called computational capability [31, 34]. The higher the value the more features of CUDA are supported on that GPU. Some properties also vary between the different underlying architectures. This means that it is important to know what kind of capability the GPU to be used has so that the program does not need features that are not there. The subjects explained here might not apply to GPUs with compute capability below 3. There are GPUs made specifically for calculations rather than games, for instance the Tesla series [33]. This section is an summary of how CUDA works mostly from the CUDA programming guide [31] and the CUDA best practices guide [34].

In CUDA the GPU is called *device* and systems CPU and memory is the *host* [31]. To perform operations on the device a type of function called *kernel* is used. A kernel is works mostly as a normal C/C++ function with the addition of some specifiers to provided options to set number of threads and so on [31]. Example 3 is an example of a simple kernel definition and example 4 shows how it can be called from the host code. The kernel add each element of the arrays A and B storing it in C. Each thread performs on addition and knows why elements to use based on it's id. The call to the kernel is made by giving three arrays and the number of threads to be used as N. In this case N needs to be the length of the arrays.

The ___global___ keyword in front of the function declaration tells nvcc that it is a kernel. The $<<< N, M >>>$ specifier tells the compiler how many N blocks and how many M threads per block that the kernel should use [31]. A block is a group of threads and shares some memory and resources. The blocks can be executed in any order so they need to be completely independent from each other but variables can be shared among threads inside a block [31]. This allows the program to scale to different GPUs as shown in figure 2.10. The blocks are organized in a one, two or three dimensional *grid*. These can be accessed by each thread so it knows which grid it is in as illustrated in figure 2.11. This can be used to make it easier to assign the threads to the correct bit of the calculation. For instance using a two dimensional grid is good for matrices [31, 34].

Example 3: Example of a declaration of a simple kernel

Data: Vectors A, B, C as C-style arrays

```
___global___ void Add(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Example 4: Host code to call the kernel in example 3 with N threads

Data: Vectors A, B, C of length N as C-style arrays

Add$<<<1,N>>>(A, B, C)$;

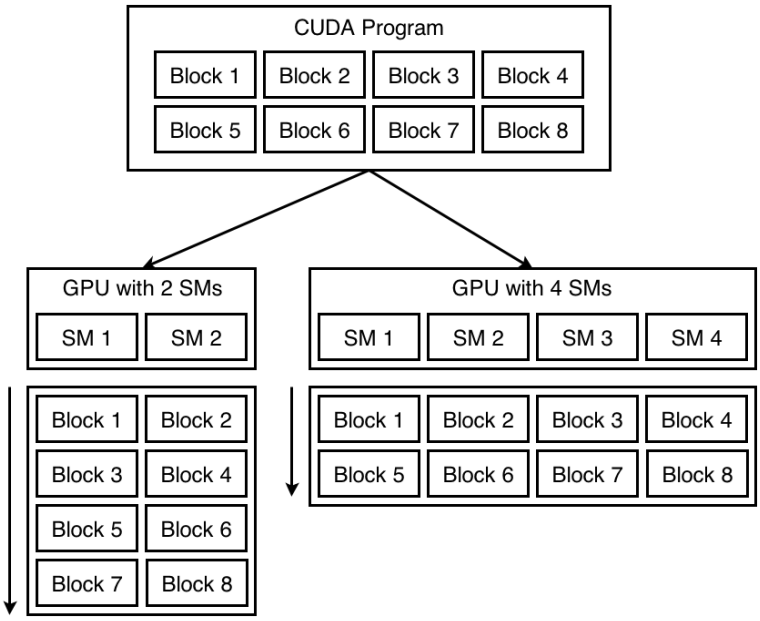


Figure 2.10: *Blocks execution depending on the number of streaming multiprocessors*

The streaming multiprocessors(SM) is the hardware that handles the execution of these blocks and can execute hundreds of threads concurrently [31]. It uses *SIMT*(Single-Instruction, Multiple-Thread) which is similar to SIMD described earlier in section 2.2.2 [31]. SIMT works mostly as SIMD except that it can act as MIMD on collections of threads called *warps* [31]. Each warp consists of 32 threads [31]. When an SM gets a block it is split into warps that are assigned to warp schedulers [31]. Each warp scheduler gives one instruction to a warp so full efficiency is achieved when all the 32 threads perform the same instruction. If there is any divergence it has to disable unrelated threads, so divergence can be costly. However different groups of warps are on different warps schedulers so can diverge without problem [31].

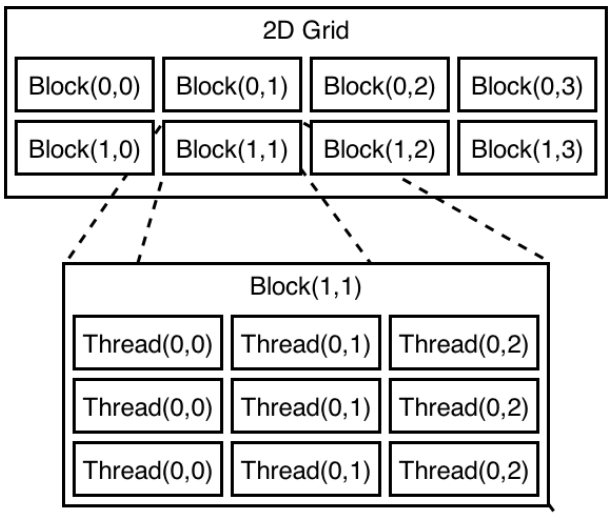


Figure 2.11: *2D grid of blocks*

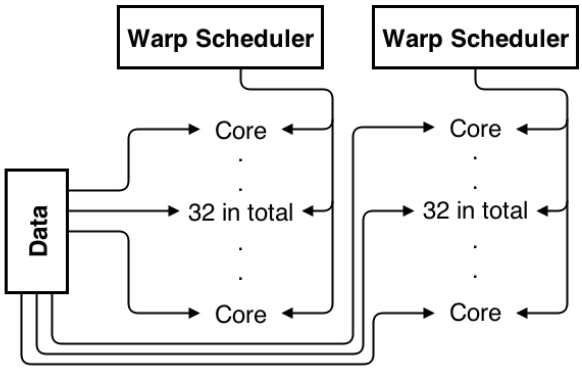


Figure 2.12: *SIMT architecture*

2.3.1 Device Memory

The GPUs memory is physically on a different device separate from the computers main memory which means that they have separate memory spaces [31]. An object in one memory is not accessible in the other memory. The computers main memory is the *host memory* and the GPUs memory is the *device memory* [31]. Since they are separate data has to be transferred to the device memory and this is done by explicit calls to transfer sections of the hosts memory [31].

The GPU also have several different types of memory [31]. Correct usage can give increased speed [31, 34].

- Register memory is located on the multiprocessor and usually costs zero cycles to access. The multiprocessor splits the available registers over its threads so if there are many threads that uses many variables not all of them will fit in the register. This is why a program sometimes can be faster with lower number of threads. [31]
- Global memory is the main memory of the GPU and is accessible from all threads and blocks. However it is relatively slow to access. [31]
- Shared memory is shared inside a block and is faster than global memory. However it is limited in size. [31]
- Constant memory is small however it is read only which enables some optimizations. It is best used for small variables that all threads access. [31]
- Local memory is tied to the threads scope, however it still resides off-chip so it has the same access time as global memory. [31]
- Texture memory is read only and can be faster to access than global memory in some situations. This was more important in older GPUs when global memory was not cached. [37, 31]
- Read-only cache is available on GPUs based on Kepler architecture and uses the same cache as the texture memory. The data as to be read only each multiprocessor can have up to 48kb of space depending on GPU. [38]

2.3.2 Streams

The kernel and transfer calls can be made on a *stream* [31]. The easiest way to think of the stream is as queue. All the kernels and transfers on the same stream will be executed in the order their calls are made, however calls from different streams can be executed in any order [31, 34]. If the kernels use a small enough amount of resources this allows up to 32 kernels to be executed concurrently(i.e. at the same time) depending on GPU [31, 34]. The transfers on the other hand can overlap even with large kernels. The overlapping of the transfers is called *synchronous transfers*. These transfers are executed on a stream and just as kernels gets executed after the previous kernels on the same stream is done. The advantage is that other streams can do calculations as normal while the transfer happens. This can hide the time for transfers completely in some situations. However the host memory that is to be transferred has to be *pinned*. Pinned memory means that the operative system can not page that piece of the memory. *Paging* means that the operative system stores a part of the memory in another area to save space in the memory, usually in the disk memory. Too much pinned memory can slow down the computer. [31, 39]

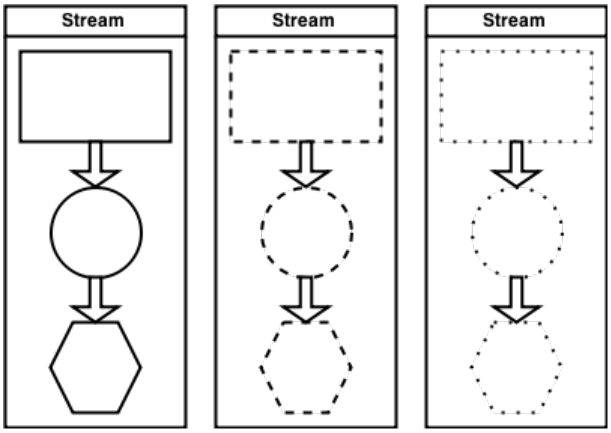
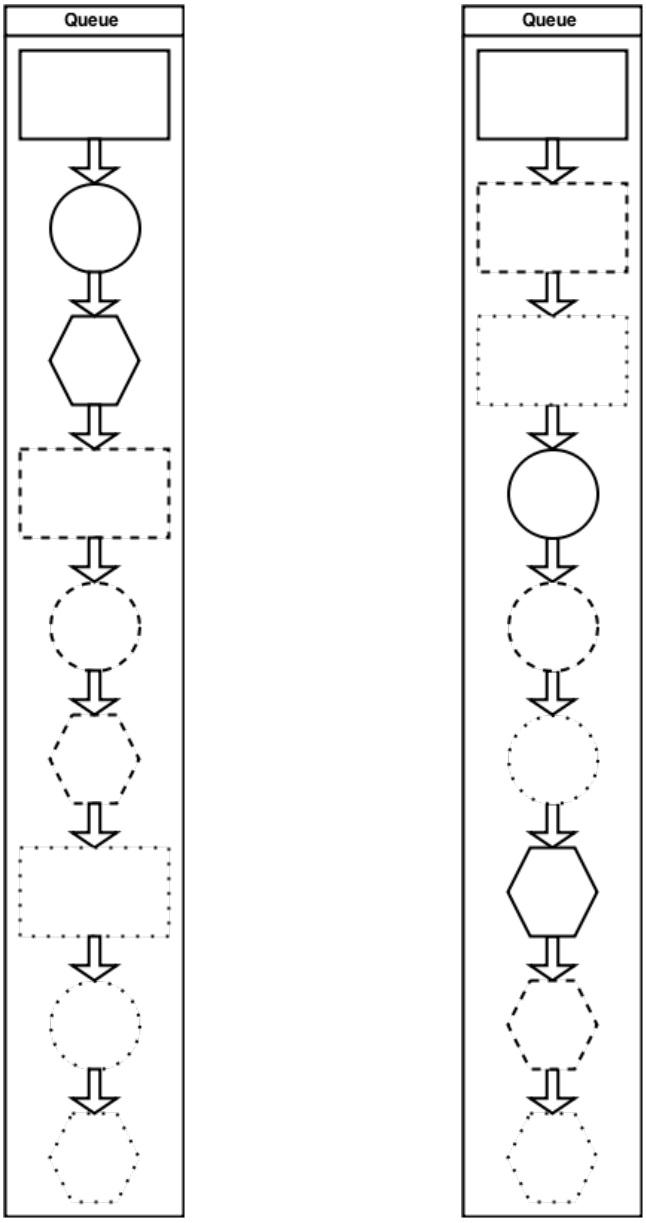


Figure 2.13: *Three kernels, rectangle, circle, hexagon, executed on each of three streams, plain, dashed, dotted*

However some GPU architectures only have one combined queue for the streams [38]. Because of this independent calls from different streams can block each other. If the queue first contains two kernels from one stream and then one kernel from another stream the GPU will execute the first kernel but not the second one until the first is complete since they are from the same stream. The GPU then does not see that the kernel on the second stream could be executed because it is blocked by the second kernel on the first stream. This affects how the kernel and transfer calls should be ordered [31, 34, 38]. Figure 2.13 shows three kernels(rectangle, circle, hexagon) on three streams(plain, dashed, dotted). If all kernels on one stream are called, then the next streams kernels then they will queue as shown in figure 2.14a. If they are performed by calling the first kernel on each stream they will queue as shown in figure 2.14b. In the former case the kernels from the first stream will block the others while in the later all the rectangle kernels could be executed concurrently [31, 34, 38]. Newer architectures from Kepler and onward has several queues so they do not have this problem [38].

The optimal ordering also depends on the number of copy engines [39, 40]. The copy engines are responsible for performing the transfers to and from the GPU. Most modern GPUs have one for each direction(i.e. to device, from device) while some of the older GPUs only have one copy engine [39, 40]. This can affect how the calls should be ordered and for GPUs with only one copy engine using the wrong order can make the performance worse than without using asynchronous transfers [39, 40].

There is an example of this in [40] which illustrates the problem. They have four versions of the same code, a sequential transfer version and three asynchronous transfer versions. Two different GPUs were used, one had one copy engine the other had two. In the asynchronous the data is split over four streams coloured differently in the figure 2.16. Version 1 initiates the calls by looping over the streams one by one and doing the transfer and kernel calls on that stream before moving on to the next [40]. Version 2 makes all the host to device transfer calls for all streams first, then the kernels and then the device to host transfer call [40]. Version 3 is the same as version 2 but with a dummy even after each kernel [40]. The figure 2.16 shows how the transfers and kernels are executed on the GPU.



(a) *Looped over stream, i.e. for each stream do all kernels on stream*

(b) *Looped over kernel, i.e for each kernel do kernel on all streams*

Figure 2.14: *Queues of the kernels in figure 2.13 called in two different ways, looped first over stream or kernel. In* a *the kernels are queued stream by stream while* b *is queued kernel by kernel.*

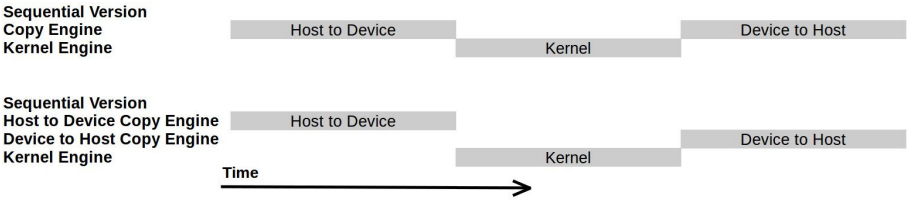


Figure 2.15: *Sequential versions*

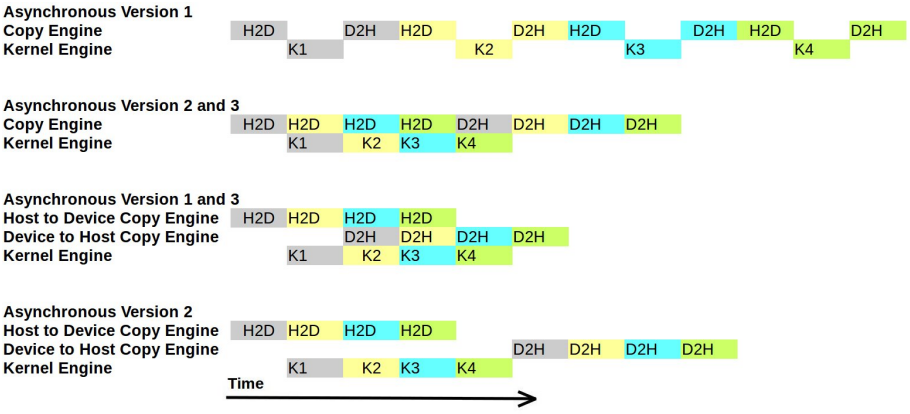


Figure 2.16: *Asynchronous versions. D2H=device to host transfer. H2D=host to device transfer. Each colour represents a stream.*

2.3.3 Efficient CUDA

One of the main criticism against GPUs for general computing purposes is that it is hard to get good performance because it requires good knowledge about details of the GPU architecture, especially the memory architecture. This section is a summary of suggestions that can be good to consider for CUDA programs from the CUDA programming guide [31], CUDA best practices guide [34] and a master thesis about GPU in GWAS [37].

Maximize parallelism
Structure the program and the algorithm in such a way that it is as parallel as possible and overlap the serial parts on CPU with calculations on the GPU. [37, 31]

Minimize transfers between host and device
Moving data between host and device is expensive and should be avoided if possible. It can be better to run serial parts on the GPU rather than moving the data to the host to do the calculation on the CPU. The bandwidth between host and device is one of the large performance bottlenecks. This can be a problem when the data is to large to fit in the relatively small GPU dram. [31, 34]

Find the optimal number of blocks and threads
There are many things affected by the number of blocks and threads so they should be considered carefully. It is a good idea to parametrize them so that they can be changed for future hardware and varied for optimization. NVIDIA has an occupancy calculator which can be helpful in determining the optimal numbers, however high occupancy does not mean high performance. [31, 34]

The number of blocks should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute. Having two blocks or more per multiprocessor can be good so that there are blocks that aren't waiting for a ___syncthreads() that can be executed. However this is not always possible due to shared memory usage and similar. [34]

The number of threads per block should be a multiplier of 32 but minimum 64. It's also important to remember that multiple concurrent blocks can reside on the same multiprocessor. Too large number of threads in a block and parts of the multiprocessor might be idle since there aren't a block small enough to use those threads. Between 128 and 256 threads is a good place to start. [34]

Use streams and asynchronous transfers
By using streams it is possible to overlap memory transfers with calculations as mentioned before. This means that the data for the next batch can be transferred while the current batch is calculated and when it is done it can start calculating on the next batch directly after the current one is done. This can hide the time for transfers completely in some situations. Depending on the time the transfers take versus the time the calculations take this can give significant speedup. [31, 39, 38]

Use the correct memory type and caches
Correct use of caches and memory is important for both CPU [28] and GPU. However it is more complicated on GPU since the caches are smaller and there are several types of memory as mentioned before [31, 34].

Avoid divergence
Each thread in a warp executes the same instruction at the same time so if some of threads diverge the rest will be ideal until they are at the same instruction again. This means it is important to use control structures such as if statements carefully to prevent threads from idling. [31, 34]

Avoid memory bank conflicts when using shared memory
Shared memory is divided into equally-sized memory modules called banks that can be accessed at the same time for higher bandwidth. Bank conflicts occur when separate threads access the same bank. On some GPUs it is fine if all threads access the same bank. Bank conflicts are split into as many conflict-free requests as needed. [31, 34]

Use existing libraries
Instead of writing everything from scratch it is usually a good idea to use already existing libraries. Especially when performance is important and most task are non trivial on GPUs so using an already optimized library is a good idea. Some of the most popular libraries for CUDA are:

- CUBLAS: BLAS implementation for CUDA. BLAS and LAPACK is a standard for a library that provides highly optimized functions for linear algebra. [41]
- CULAtools: BLAS and LAPACK implementation for CUDA for both dense and sparse matrices [42]
- MAGMA: BLAS and LAPACK implementation among other things that can distribute the work on both CPU and GPU [43]
- Thrust: Template based library that tries to emulate C++ standard library [44]

Avoid slow instructions
There are some instructions that can be slow and should be avoided if possible, for instance type conversion, integer division and modulo. If a function is called with a floating point number that might be used as a double and require a conversion. By putting an f at the end of the number it is told to be single precision float, for instance 0.5f. In some cases it is possible to use bitwise operations instead which is faster. [31, 34]

Restricted pointers can give increased performance
Aliasing can be a problem as mentioned earlier. By using the ___restrict___ keyword on pointers the compiler can be told that no aliasing will occur, however it is up to the programmer to make sure that is the case or there might be unexpected results. Not using aliasing reduces the number of memory accesses the CPU needs to make. However it increases register pressure so it can have an negative effect on performance. [31]

Use fast math functions if precision isn't needed

There are three versions of the math functions. The double precision version is func() while the single precision function is funcf(). There is third faster but less accurate version _funcf(). The option -use_fast_math makes the compiler change all the funcf() to _funcf(). [34]

Instruction level parallelism can increase speed

Just as for CPUs the GPUs can take advantage of instruction level parallelism. By unrolling loops this can give 2x the speed relatively simple [45].

2.4 Software Design

How to design software so it can be maintained over time has been a problem for a long time [46, 47]. Object oriented languages such as C and later Java and C++ arose because of problems with maintaining software [46].Badly organized and written code will cost productivity in the future when bugs and other problems stack up because of earlier mistakes [46, 47]. Correcting those bugs are likely to be time consuming because it can be hard to find where they originated from. All code gather problems overtime, however a well designed system will degenerate significantly slower than one that was designed carelessly [46]. The code will also be read by ourselves and others later which means readability is important if a future reader is to understand the code and find the parts they are interested in [46].

This is also important in science where others might wish to use the tools, repeat the experiments or build upon it. A study [48] found that the repeatability in computer science is low. Their goal was to get the code from articles in computer science and compile it, meaning that they did not test if the code actually ran properly or even at all. Their results are shown in 2.17. Of 515 articles they received the code from 231, of those 102 compiled [48].

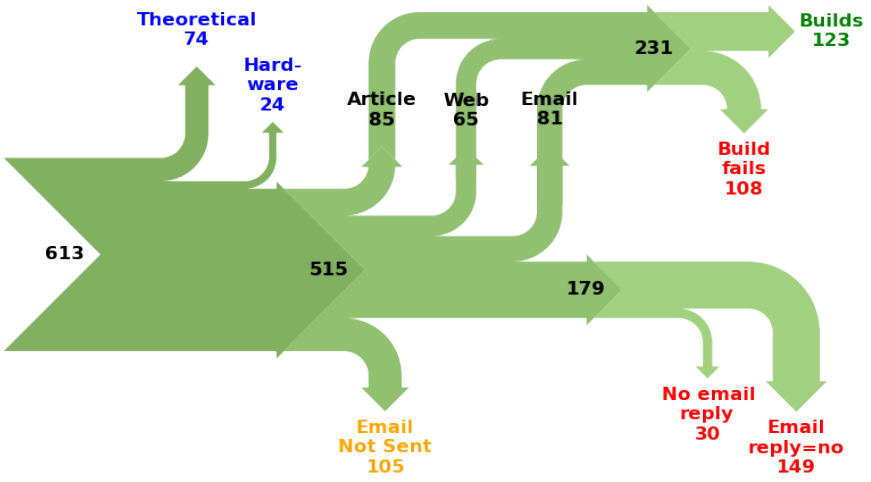


Figure 2.17: *Repeatability in computer science,from [48]*

There are ways to design the program and code in such a way that it is easier to read and maintain. Most of the concepts described here goes under the development technique called agile development [46]. SOLID is an acronym for five principles for object oriented programming and design [46]. When used together they are intended to make programs that are easy to maintain and extend [46]. As already mentioned readability is important. Correct names will make the code explain itself without other documentation(e.g. comments), the code itself is the documentation [46].

Initial	Principal	Concept
S	Single responsibility principle	A class should only have a single responsibility
O	Open/closed principle	A class should be open for extension but closed for modification
L	Liskov substitution principle	If S is a subtype of T, then objects of type T may be replaced by objects of type S without without altering any of the desirable properties of the program(e.g. work performed)
I	Interface segregation principle	Use several smaller and more specific interfaces instead of one large
D	Dependency inversion principle	Depend on abstractions(e.g. interfaces) not details

Table 2.4: *The five SOLID principles, from [46]*

Dependency injection is one way to implement dependency inversion principle [46]. Injection is passing the dependency to the dependent object. This is used instead of allowing the dependent object to construct or find the dependency [46]. *Factories* are used to allow for dependency injection when the class has some function that creates new instances of the class [46]. Since the class creates them they can not be directly replace so instead the construction is delegated to a factory and the factory class can be used with dependency injection [46]. There are also other uses for factories such as eliminating code repetition from construction of classes that are constructed from a chain of classes(e.g. a house consists of windows, doors, walls, etc) or when there are several classes that inherit from the same base class [46, 47].

2.4.1 Unit Tests and Mocks

Unit testing involves testing the program in small units in isolation [46]. Testing in isolation means that the test should only depend on the part of the program that is tested [46]. If a part of the program is not working properly only its related tests should fail, not other tests for code that depends on it but otherwise works properly [46]. This makes it easier to find errors when they do occur since the tests will pinpoint the unit which does not work. *Integration testing* is used to test several units together to find possible errors that occur when the smaller units are combined.

It can be easy to denote test code as less important than the main code but they should be treated as equally important [46]. Tests should also not be an inconvenience to use so they should be easy to run, take reasonable amount of time to complete and not require any outside interpretation whether they failed or not [46].

Mocking is to replace a real object with a fake object called a *mock* that for the code is indistinguishable from the real object [46]. This allows one to create situations and test with more control and without depending on the real objects code [46]. The second is important for unit tests since it enables one to test units that normally depend on others in isolation [46]. In the first case it is useful in situations such as when one wants to test a class handling of a rare failure [46]. Such failures can be hard and time consuming to induce. It is then easier to use a mock object that behaves like the failure has occurred. However mocking requires that the code for the class doesn't create the object itself since there is no way to replace the object with the mock [46]. This is one of the reasons why dependency injection should be use [46].

2.4.2 Design Patterns

A *design pattern* in software design is a reusable general solution to a common recurring problem in a given context [47]. It is templates and structures to solve the problem, however it is not code [47]. However they are partially dependent on the programming language since different languages have different features and limitations [47].

Consumer producer is a concurrency pattern for when there are a number of consumers/workers and producers [46]. They share a common queue for products. The producer generates some product and put it into the queue, while the the consumers consume the products. The problem is that the producers should not add to an non empty slot and that each product should only be consumed once [46, 47].

An *wrapper* is an interface used to *wrap* another interface, it is also called *adapter* pattern [46, 47]. The wrapper does not perform the work, it delegates it to the other interface [46]. This is useful for instance when a 3rd party interface is not in the same style as the program or has other problems but otherwise suits the needs [46].

2.4.3 Version Control Software

Backups for data, code, text and so on is important if something happens. One way to backup text and code projects is to use a version control software and is usually an important tool in developing [49]. For this project Git was used and the source code can be found at `https://github.com/Berjiz/CuEira`. Version control software allows several developers to share a common *repository* which allows them to work on different parts at the same time, it could even be in the same file [49]. However changes at the same place causes *merge conflicts* which usually needs to be solved manually [49]. A version control software keeps track of all the changes made so if a part later turns out to be wrong that part can reverted to an older correct version [49]. Version control software also have many other features [49].

2.5 Performance Measures

An important part in creating fast and efficient programs is to know how fast the program is under certain conditions and which parts of the program are slow [23]. For instance the speed could suddenly drop when too many threads, there might be a bottleneck in the communication, and so on.

There are two ways to measure how long a program takes to execute [23]. Wall clock time is how long real life time the program took [23]. The other is to measure the number of processor cycles spent [23]. A parallel program will have shorter execution time than it is serial version however it will likely have spent more processor cycles due to overhead from communication and initialisation of the threads. These two measures are useful for different kinds of comparisons. Wall clock time is better for overall performance while number of cycles is useful for comparing different algorithms [23, 34].

Speed up is a measure of how much faster then program is with a certain number of threads compared to the serial version. It's defined as [23]

$$S(p) = \frac{T(1)}{T(p)} \tag{2.11}$$

Where $T(1)$ is execution time of serial program and $T(p)$ is execution time of parallel program with p threads. Linear speedup is when S(p)=p [23].

Efficiency reflects how efficient the program is using p threads. Linear speed has efficiency 1. It's defined as [23]

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} \tag{2.12}$$

Strong scaling refers to how the program handles a fixed problem size and increased number of processors [23]. An program with strong scaling has linear speedup [23]. Weak scaling refers to the execution time of the program when there is a fixed problem size *per processor* and the number of processors is increased [23, 34].

It can be a good idea to plot these measures while varying p, this can show when a bottleneck occurs. Looking at the measures at node level can also be useful to get an idea of how increased number of nodes and therefore increased communication over the network affects the performance.

2.5.1 Amdahl's Law and Gustafson's Law

Amdahl's Law is used to find the expected speedup of a system when parts of it are made concurrent [50]. Simply it says that as the number of processors increases the parts that aren't parallel will start taking up more and more of the wall clock time and that the speedup for adding more processors will decrease as more and more processors are added and more time is spent relatively on the non parallel parts [23, 50]. It's closely related to strong scaling [34, 50].

It says that the expected speedup with F fraction of the code parallel and p threads is [23]

$$S(p) = \frac{1}{(1-F) + \frac{F}{p}(1-F)} \tag{2.13}$$

As the number of threads grow towards infinity $S(p)$ converges on $\frac{1}{1-F}$. If we have 90% of a code parallel then even with infinite number of threads we won't get a better speedup than ten [50]. There are limitation to Amdahl's Law since it makes a couple of assumptions [50, 51].

- The number of executing threads remain constant over the course of the program.
- The parallel portion has perfect speedup. Often not true due to shared resources,eg caches, memory bandwidth, and shared data.
- The parallel portion has infinite scaling, not true due to similar limits as above. More threads will not increase performance after a while or might even decrease it.
- There is no overhead for creation and destruction of threads.
- The length of the serial portion is independent of the number of threads. Often the serial work is to divided the work to the threads, this work will obviously increase as the number of threads go up. More threads can also lead to more communication overhead.
- The serial portion can't be overlapped by the parallel parts. For instance with producer consumer type pattern the consumer could be strictly serial but the time it takes could be overlapped by the parallel producers.

This means it is most accurate with programs that are of the fork-join type, e.g. both serial and parallel parts [51].

Gustafson's Law is closely related to Amdahl's Law and can in some ways be more accurate than Amdahl's Law [51]. Gustafson's Law makes similar assumptions as Amdahl's Law however it also makes two additional statements. It states that problems tends to expand when provided with more computational power, e.g. increased precision by reducing grid size for simulations, higher frame rate for graphics and so on [51, 23]. The second is that the parallel portion of the program tends to expand faster than the serial part, e.g. for matrix multiplication the initialisation scales linearly with the matrix size while the multiplication itself scales as $O(n^3)$ [51, 23]. The former means that it is

closely related to weak scaling [23]. So in a way it says that the execution time remains constant rather than the amount of data. More precise it says that the expected speedup with p threads and F fraction of the code that is parallel is [51, 34, 23]

$$S(p) = p + (1 - F)(1 - p) \tag{2.14}$$

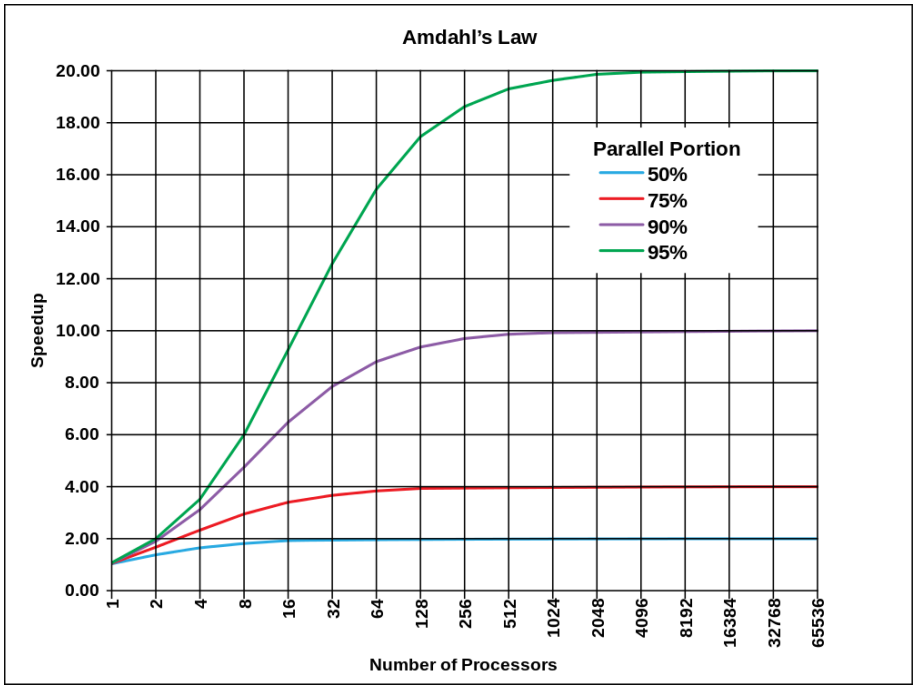


Figure 2.18: *Illustration of Amdahl's Law. Wikipedia Commons*

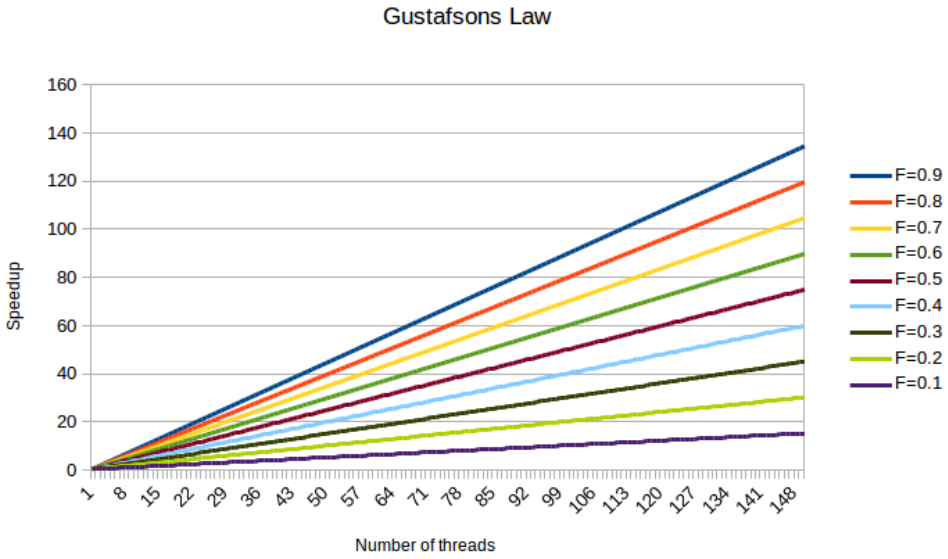


Figure 2.19: *Illustration of Gustafson's Law*

2.5.2 Profilers

There are applications called profilers that are made to assess the programs performance and resource consumption [23, 34]. They calculate some of the measures mentioned earlier and they also check hardware usage and how much time the program spends at various parts of the program [23, 34]. This is very useful for finding bottlenecks and other problems in the program. It does not matter if the algorithm is super fast if all the data is stuck in network transfers. The profilers can be hardware dependent so the manufactures usually provided them for their products [23, 34].

Nsight [52] is a combined profiler and integrated development environment(IDE) based on either Visual Studio or Eclipse. For this thesis Nsight Eclipse edition was used. It works as the usual Eclipse but has added functionality for CUDA and CUDA profiling [52].

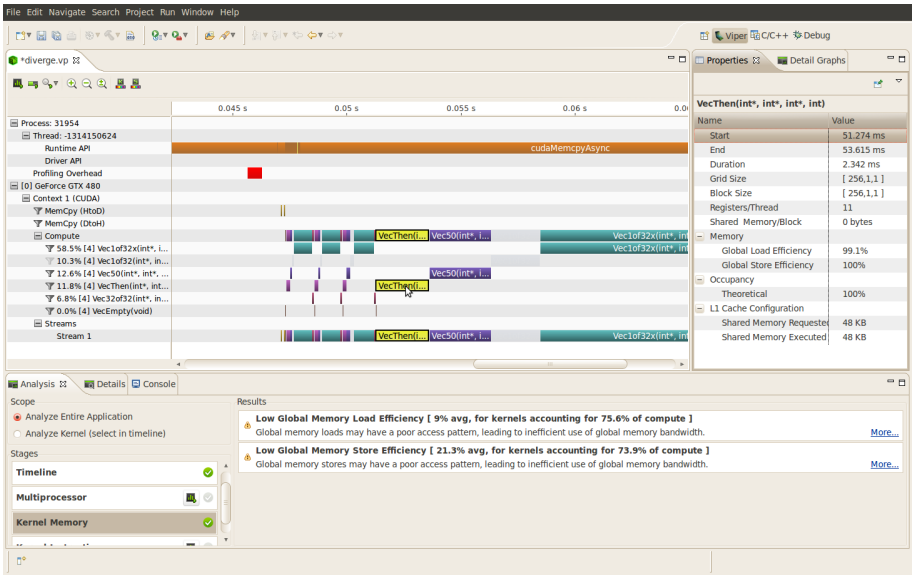


Figure 2.20: *Screenshot of Nsight Eclipse Editions profiling section, from [52]*

2.6 Algorithms

There are many algorithms and programs proposed for searching for interaction and most have focused on gene-gene interaction [2]. One of the challenges of gene-environment interaction is that environmental factors can be of any variable type(binary, continuous, categorical) which creates problems in various ways [2]. Gene-gene interaction tools can sometimes be used to find gene-environment interaction, however they usually require the variables to be binary or have other problems since they were not designed for environmental interaction [2].

Both clusters using only CPUs [53] and CPUs together with GPUs [8, 54, 55, 56, 57, 37, 58] have been used for gene-gene interaction. SNPsyn also used Xeon Phi, they got 1.5 to 2 times speedup with a K20 GPU compared to the Xeon Phi [58]. The studies got high gains from implementing their algorithms on GPU compared to their CPU versions [8, 54, 55, 56, 57, 37]. However most studies CPU versions were not parallel so it is likely that the gains would have been smaller if they had compared to an optimized and parallel CPU version. One study made a comparison of their algorithm between using a CPU cluster and using a GPU [59]. They found that 16 CPU nodes had the same performance as a single GTX 280 card, however the study is from 2009 [59].

The methods can be roughly classified into four categories, exhaustive, stochastic, machine learning/data mining and stepwise [10].

Exhaustive search is the most direct approach, it compares all combinations of the SNPs in the dataset. Exhaustive search methods will not miss a significant combination because it didn't consider that specific combination. However it also means that they can be slow since they will spend time on combinations other methods would skip completely. Multifactor-Dimensionality Reduction(MDR) [60] and BOOST [61] are two examples of this type of algorithm.

Stochastic methods uses random sampling to iterate through the data. BEAM [62] is one example and it uses Markov Chain Monte Carlo(MCMC) method.

Data Mining and *Machine Learning* are methods that try to learn patterns from data and tries to generalize it. MDR [60] is a type of data mining method and is among the most common methods used in GWAS. See section 2.6.2 for more details.

Stepwise methods uses a filtering stage and a search stage. At the filtering stage uninteresting combinations are filtered out by using some exhaustive method. The other SNPs are the examined more carefully in the search stage. BOOST [61] is an example which uses succinct data structures and a likelihood ratio test to filter the data before applying log-linear models.

2.6.1 Logistic Regression

One way to model the contingency tables is by using *logistic regression*. Logistic regression is a type of linear regression model for classification that models a latent probability for the outcomes [17]. The outcomes are binary, however the method can be extended to multiple outcomes. In this work we will only consider them as binary. Logistic regression transforms the probability by using the *logit* transformation [17]. The logit transformation with probability π is [17]

$$\log(\frac{\pi}{1-\pi}) \tag{2.15}$$

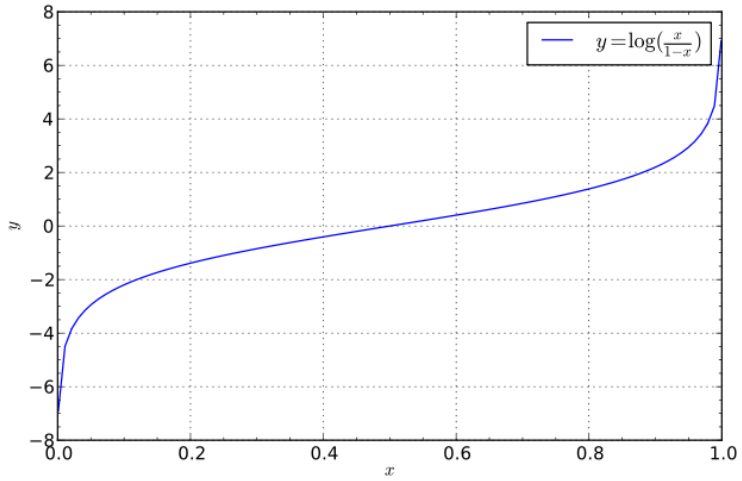


Figure 2.21: *Graph of the logit transformation. Wikipedia Commons*

The probability with a set of predictor variables X is $\pi(X) = P(Y = 1)$ [17]. The linear regression model with n predictors $X = (x_1, x_2, ..., x_n)$, coefficients $\beta = (\beta_1, \beta_2, ..., \beta_n)$ and by using the logit transformation is then [17]

$$logit[\pi(X)] = \alpha + \beta X \tag{2.16}$$

By moving the logit to the right side of the equation we get the model of the probability [17]

$$\pi(X) = \frac{e^{\alpha+\beta X}}{1 + e^{\alpha+\beta X}} \tag{2.17}$$

The logit, equation 2.15, also happens to be the log of the odds(equation 2.5) [17]. By exponentiating both sides of equation 2.16 it shows that the odds is [17]

$$e^{logit[\pi(X)]} = \frac{\pi(X)}{1 - \pi(X)} = e^{\alpha+\beta X} = \Omega \tag{2.18}$$

This means that $\exp\beta$ is the odds ratio since the odds increase by $\exp\beta$ for each unit increase of X [17]. It can also been seen by taking the ratio of the odds using equation 2.6 and with $X = x + 1$ and $X = x$

$$\theta = \frac{e^{\alpha+\beta(x+1)}}{e^{\alpha+\beta x}} = e^{\alpha+\beta(x+1)-\alpha-\beta x} = e^{\beta} \tag{2.19}$$

Finding the β coefficients are done in a similar way as with other linear regression models since they all are generalized linear models [17]. It's usually done using maximum likelihood(ML), via Newtons method [17, 12]. It's an iterative method which means it can be relatively slow compared to non iterative methods. The pseudo code for the algorithm using Newtons method can be found in algorithm 3 [12]. Line 13 is sometimes inside the loop, however since it is not needed for the actual iteration calculating it for every loop wastes time.

Algorithm 3: Logistic regression using maximum likelihood and Newtons method

Data: N number of data points
M number of variables, excluding the intercept
$\boldsymbol{X}$ is a $N \times M$ matrix that contains the variables
$\boldsymbol{Y}$ is the outcomes with length N
β has length M and contains the β coefficients
p has length N and contains the probability for the outcome for each individual
s has length M and contains the scores of the coefficients
$\boldsymbol{J}$ is a $M \times M$ matrix called the Jacobian, also called information matrix
Note: $*$ is element by element multiplication

1 $\boldsymbol{X} \longleftarrow \begin{pmatrix} 1 & \\ \vdots & \boldsymbol{X} \\ 1 & \end{pmatrix}$

2 $\beta \longleftarrow \begin{pmatrix} 0 \\ \beta \end{pmatrix}$

3 $iter \longleftarrow 0$
4 $diff \longleftarrow 1$

5 **while** $iter < max\_iter$ ***and*** $diff > threshold$ **do**
6 $\quad \beta_{old} \longleftarrow \beta$
7 $\quad p \longleftarrow \frac{e^{\boldsymbol{X} \cdot \beta}}{1+e^{\boldsymbol{X} \cdot \beta}}$
8 $\quad s \longleftarrow \boldsymbol{X^T} \cdot (\boldsymbol{Y} - p)$
9 $\quad \boldsymbol{J} \longleftarrow (\boldsymbol{X^T} \cdot (p * (1-p))) \cdot \boldsymbol{X}$
10 $\quad \beta \longleftarrow \beta_{old} + \boldsymbol{J}^{-1} \cdot s$
11 $\quad diff \longleftarrow \sum |\beta - \beta_{old}|$
12 $\quad iter \longleftarrow iter + 1$

13 $log\ likelihood \longleftarrow \sum(\boldsymbol{Y} * lnp + (1 - \boldsymbol{Y}) * ln(1-p))$

2.6.1.1 Matrix Inverse and Matrix Decomposition

The inverse of the *information matrix*, **J**, on line 10 in algorithm 3 is generally not possible to do with normal matrix inversion because it is not defined for a general matrix [63]. However the *pseudoinverse* is defined for a general matrix, it is denoted as A^+ for the matrix A [63, 64]. There are several types of pseudoinverse, one of the more common is the *Moore-Penrose pseudoinverse* where A^+ is the matrix that satisfies equations 2.20 to 2.23 [63].

$$AA^+A = A \tag{2.20}$$

$$A^+AA^+ = A^+ \tag{2.21}$$

$$(AA^+)^T = AA^+ \tag{2.22}$$

$$(A^+A)^T = A^+A \tag{2.23}$$

Matrix decomposition, also called *matrix factorisation*, are methods to factorize a matrix into products of matrices [64]. Some of the decomposition methods can be used to find the pseudoinverse. *Singular value decomposition*(SVD) is one of them and works for general matrices [64]. It's factorization of a $n \times m$ matrix A is shown in equation 2.24 [64].

$$A = U\Sigma V^T \tag{2.24}$$

Where Σ is a diagonal matrix with non negative numbers, its diagonal values are the *singular values* of A [64]. Using SVD the pseudoinverse is shown in equation 2.25 [64]. Σ^+ is the pseudoinverse of Σ. This is done by inverting each non zero element of the diagonal and transposing the matrix [64].

$$A^+ = V\Sigma^+U^T \tag{2.25}$$

2.6.2 Data Mining and Machine Learning Approaches

Approaches based on Data Mining and Machine Learning have been a popular choice for GWAS. Random Forest(RF) [65] and MDR [60] are among the most common ones [2, 1]. There are other methods as well such as clustering approaches [10]. Most of them are used for filtering the data for possible interactions and using another method after the filtering [2, 1].

One of their larger advantage is that they are usually non-parametric and designed with high dimensional data in mind [1]. However they are prone to overfitting and the usual way to try to prevent that is to use cross validation or permutation tests [1]. It means that even if the method itself is fast the algorithm is repeated so many times that the whole algorithm can be slow in the end [1].

2.6.2.1 Random Forest

RF is an ensemble learning method [65]. Ensemble methods combine multiple models to improve performance. RF takes randomized samples of the data and builds decision trees on each of them [65]. These trees are then combined to form the classifier. Usually hundreds or thousands of trees are used depending on the problem [65]. One of the most popular variants of Random Forest for GWAS is Random Jungle [66].

It has been shown in high dimensional data that RF tends to only rank interacting factors high if they have strong marginal effects [67]. Also the ranking of the variables does not indicate which factor it is interacting with either since it is based on the joint distributions [2]. How to incorporate the environmental factors in RF is also not obvious and using variables with very different scales can bias the results [2].

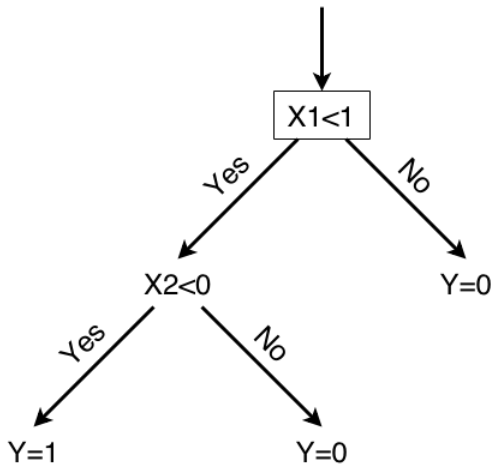


Figure 2.22: *An example of a decision tree*

2.6.2.2 Multifactor-Dimensionality Reduction

MDR is a method that reduces the number of dimensions(i.e. variable) by combining several dimensions into one [60]. In GWAS it combines the variables that are suspected to interact. This new variable is then compared against the outcome [60]. If the new variables predictability of the outcome is high enough then the variables that were combined are considered to interact [60]. This process is usually repeated on all pair combinations of variables [1, 60].

The reduction from n dimensions is done by calculating the ratio of cases versus controls for each combination of the possible values of the variables [60]. If the ratio is above a certain threshold all the members of that group get the value 1 for the new dimension, otherwise 0 [60]. Accuracy of the model is measured by using cross validation and permutation tests, in other words it reshuffles the data randomly and recalculates the model a large number of times to get an estimate of the models certainty [1, 60]. Because of that MDR can be slow [1, 60]. However it is still usually faster than exhaustive search with regression methods [1, 60].

MDR can been used for gene-environment interaction but requires modifications since MDR can only handle binary variables [2]. There are extensions that can use continues variables, however these are regression based so these will be slower than regular MDR [2].

A simple example of MDR using exclusive or (XOR). XOR is an logical operator that is true if one and only one of its two variables is true. We have 4 possible combinations and an occurrence for each of them, see table 2.5. The combination (1,0) and (0,1) both have one case with outcome 1 so MDR will classify them as 1 in the new variable Z, the other two combinations have outcome 0 so will be classified with Z=0, see table 2.6. From here it is easy to make an predictor from Z to the outcome Y by comparing the values.

$\mathbf{Y}$	$\mathbf{X_1}$	$\mathbf{X_2}$
1	1	0
1	0	1
0	0	0
0	1	1

Table 2.5: *XOR table with outcome* $\mathbf{Y}$ *and variables* $\mathbf{X_1}$ *and* $\mathbf{X_2}$.

$\mathbf{Y}$	$\mathbf{Z}$
1	1
1	1
0	0
0	0

Table 2.6: *XOR table with* $\mathbf{X_1}$ *and* $\mathbf{X_2}$ *combined into* $\mathbf{Z}$ *using MDR.*

Chapter 3

Implementation

This chapter will explain the implementation of the program starting with a short summary of how the older programs, JEIRA and GEISA, work. After that, a more thorough look at the structure and implementation of CuEira. The input files for all these programs are PLINK data files. However CuEira can only read the PLINK files in binary format. These programs also read a separate tab delimited file that contains the environmental data and covariates if any. For CuEira the covariates and environmental data are in two separate tab delimited files.

3.1 JEIRA and GEISA

The descriptions and structures in this section applies to both JEIRA and GEISA since their underlying structure is the same, however the focus is on GEISA. Both programs are written in Java and uses Javas features for concurrency.

GEISA has three dependencies, two are provided as JAR files in the distribution so the user doesn't need to get install them separately. The dependency Apache Maven is only needed if the user wants to compile the program from source.

Apache Commons CLI
: Provides command line interface, e.g. it parses the options when using the program.

Apache Commons Math
: Provides matrix and vector classes, linear algebra, etc.

Apache Maven
: Used to compile Java programs.

The implementation uses the producer consumer pattern described in section 2.4.2. The main thread creates a queue of tasks which all of the result producers iterate over and outputs results. These results are placed in a queue and consumed by another thread that writes them to a file. The program reads all data at the start which means it can get memory problems for large datasets. JEIRA and

GEISA are both largely without tests, however there are a few unit tests for some of the basic storage classes and the binary file reader.

3.2 CuEira

This section will explain the structure of CuEira and how it works starting with a general overview. Later the parts are explained and finally how the parts are combined to form the program. C++ was chosen because it is generally fast and have good control of the memory. The downside is that it is more platform dependent than Java. Java can also be easier to work with in several ways.

The basis of the program is that each SNP is handled independently and a SNPs data is only read from the file when its needed. This reduces memory usage since only a few SNPs are stored in the memory at the same time. A number of worker threads share a queue of the SNPs and with each worker thread corresponding to a stream on a GPU. The program is split into two stages, initialisation and calculations. The initialisation part is done by the main thread and reads all the information except the SNP data. In the calculation step each worker threads fetches a SNP from the queue, reads its data, calculates the model, writes the results and then starts again with the next SNP until the queue is empty.

An alternative solution would be to use *callbacks* to control the execution of the kernels. A callback is a function in CUDA which after a kernel or transfer is complete starts a thread that calls a global function [31]. These new threads can not call new kernels or transfers because it could create infinite loops [31]. However the callbacks can be used to change a state tracked by the main thread which then tells the thread to execute the next kernel or transfer. This solution was not used because it is more complex without providing any significant advantages.

Copying of the classes is avoided to save performance. Most of the classes have the copy and move constructor/operator deleted to prevent accidental usage. Accidental usage would likely cause errors because the default copy and move constructor/operator are naive and this will causes problems with more complex classes [68].

Factories are used at several places to help satisfy the dependency inversion principle. Without the factories some objects can not be mocked and therefore a proper unit test can not be performed.

To improve the codes clarity several things were done. There are several namespaces to separate different parts of the program and to reduce clutter. For the same reason all classes have appropriate names and abbreviations were avoided in most of the cases. Comments were used sparsely because the names should be clear enough and explain what the code does. Comments are also easy to forget to update when the code is changed.

There are timers at various places that can be used to profile parts of the program. Some of them count how much time is spent waiting at locks or specific parts of the calculations.

3.2.1 Dependency and Compiling

CuEira have some dependencies. The dependency on MKL and Intels compiler is something that can be changed. BLAS shares the interface so another BLAS library could work with updates to the CMake file.

Make
Used for compiling

CMake
Used to create Make files

Boost
Boost is a collections of C++ libraries. For CuEira it is used for file handling, string operations and parsing the command line options among others.

CUDA
NVIDIAs library for GPU, explained in section 2.3

CUBLAS
BLAS for CUDA

MKL
Intels BLAS library

Intel Compiler
Intels C/C++ compiler

Google Test and Google Mock
Unit test and mocking framework. They are provided in the CuEira source so the user does not need to install them separably.

CuEira is compiled by using *CMake* and *Make*. CMake generates Make files from a CMakeLists file. Make files sets rules for how the program should be compiled. Compiling can be complicated and using CMake is one way to make it easier. One of the main advantages of CMake is the modules and the command *find package* which can find libraries by searching for them using common paths. The modules can also help with linking the library with the program and can sometimes also provided other functions. For instance the find package for CUDA has commands to compile CUDA code. There is also support for cross platform compiling which is something that could be used in the future to compile CuEira for other operative systems than Linux.

The main CMake file needs to know what and where all the source files are. This is done by using nested CMake files, each directory has an CMakeLists file that adds it's sub directories and sets all the source files to the previous level. There are four types of lists of source files each is compiled into a library and then linked together. There is one list for all the tests, one for the files that needs to be compiled with nvcc, one for other GPU related code and then one with the host code.

The CMake file contains several options that controls how CuEira is compiled. For instance one of them sets if it should be single or double precision. More details on how to compile and use CuEira can be found in appendix E.

3.2.2 Storage

A set of container classes holds the data. There are two types of them, the basic matrix and vector classes and data containers. The data containers are responsible for handling the data for the model and performing the recoding. They use the vector classes for the actual storage.

There are three types of matrix and vector classes which are all stored in column major format. The set with the Device prefix is for storage on the device and the other two are for storage on the host and shares a common interface. The difference between the host sets is that the ones with Pinned prefix uses pinned memory while Regular does not.

Other basic information such as the information about individuals, environmental factors and SNPs are stored in simple classes where each individual, environmental factor or SNP is a single object. *Enums* are used to store some of them.

The SNPs are stored in the class **DataQueue**. **DataQueue** provides the function **next** that fetches a SNP from the queue and returns it. It has a lock so it can be shared among all worker threads.

The **EnvironmentFactorHandler** is responsible for keeping track of the **EnvironmentFactors** and it's corresponding data. When constructed it updates the EnvironmentFactors variable type. This is done because it can sometimes be needed to know if a variable is binary or not. It also provides functions to access the **EnvironmentFactors** and the corresponding vectors.

There are three data container classes, **SNPVector**, **EnvironmentVector** and **InteractionVector**. They all have the responsibility to keep track of the recoded data and to perform the recoding when their **recode** function is used. The **recode** function takes an **Recode** enum which tells it which type of recoding it should perform. The **SNPVector** also holds the original SNP data. **EnvironmentVector** doesn't need to since **EnvironmentFactorHandler** has that responsibility and **InteractionVector** does not have any kind of original data.

The **DataHandler** is responsible for keeping track of the current combination of data containers being used and to iterate to the next one when its **next** function is called. When the **next** function is used a couple of things happen. If all environmental factors have been calculated together with the current SNP it deletes the SNPs information, asks the queue for a new one and reads its data and switches to the first environmental factor. Otherwise it moves on to the next environmental factor. It then updates the InteractionVector and the Statistics classes to create the contingency table and the allele frequencies. After that the next set of variables is ready to be used.

3.2.3 File Input and Output

CuEira reads five files which are in four different formats. There are five classes for reading the files, one for each file.

Three of the files are PLINK files in binary format. The fam file has the information about the individuals, the bim file has the names of the SNPs and their alleles. The genetic information for each individual is in the binary bed file. More details on PLINK files can be found in the appendix B.1. The classes used to read them are **BimReader**, **BedReader** and **FamReader**.

The other two files are CSV files. One of them contains the covariates and the other the environmental factors. **CSVReader** is responsible for reading them and storing the data in a matrix. The fifth reader class is **EnvironmentCSVReader** which inherits **CSVReader**. Its purpose is to convert the matrix read by **CSVReader** with the environmental data to **EnvironmentFactorHandler**.

All the information except the genetic data in the bed file is read during the programs initialisation. When a new SNP is being used for calculations its data is read from the bed file. Most of the memory needed to store the data is in the bed file so only reading it when needed save a lot of memory.

There is only one writer, the **ResultWriter**, and as its name suggests its responsibility is to write the results. The results are written immediately and then discarded to save memory. There is an lock on the writing so several threads can share it. The results will therefore be written in the order they are completed. The format of the output file is described in appendix B.

3.2.4 Initialisation of the Variables, Recoding and Statistic Model

The initialisation of the **SNPVector** and **EnvironmentVector** is done by calling the recoding with ALL_RISK from the **Recode** enum. The **recode** function calls different functions based on the enum. If it is ALL_RISK it calls a function that copies the data from the vector holding the original data to vector that holds the data that will be used for the model. **InteractionVector** does recoding and initialisation the same way because in both cases the interaction data is the multiplication of the elements of the **SNPVector** and **EnvironmentVector**. The algorithms for the recode functions are shown in algorithm 4, 5 and 6.

For the additive statistical model all the elements in **SNPVector** and **EnvironmentVector** need to be set to 0 when the corresponding element in **InteractionVector** is 1. Both classes have an function named **applyStatisticModel** that performs it. The function takes the **InteractionVector** as an parameter and the algorithm is the same for both **SNPVector** and **EnvironmentVector**. The function is shown in algorithm 7.

DataHandler has the overall responsibility for recoding and applying the statistical model, it calls the data containers functions as needed. It also updates the **ContingencyTable** after recoding since the distribution of the groups changes when recoding.

Algorithm 4: SNPVector protective recoding

Data: Two vectors of length N, $originalData$ and $recodedData$
$snpData0$, $snpData1$, $snpData2$ stores the value for the final vector that corresponds to the original vectors values of 0,1 and 2
i is an index from 0 to $N-1$

```
if Dominant genetic model then
    if Risk allele is allele 1 then
        snpData0 ← 1
        snpData1 ← 1
        snpData2 ← 0
    else if Risk allele is allele 2 then
        snpData0 ← 0
        snpData1 ← 1
        snpData2 ← 1
else if Recessive genetic model then
    if Risk allele is allele 1 then
        snpData0 ← 1
        snpData1 ← 0
        snpData2 ← 0
    else if Risk allele is allele 2 then
        snpData0 ← 0
        snpData1 ← 0
        snpData2 ← 1

for i ← 0 to N do
    if originalData[i] = 0 then
        recodedData[i] = snpData0
    else if originalData[i] = 1 then
        recodedData[i] = snpData1
    else if originalData[i] = 2 then
        recodedData[i] = snpData2
```

Algorithm 5: EnvironmentVector protective recoding

Data: Two vectors of length N, $originalData$ and $recodedData$
i is an index from 0 to $N - 1$

for $i \leftarrow 0$ **to** N **do**
  if $originalData[i] == 0$ **then**
    $recodedData[i] = 1$
  else
    $recodedData[i] = 0$

Algorithm 6: InteractionVector recoding

Data: Three vectors of length N, $interactionData$, $envData$ and $snpData$
i is an index from 0 to $N - 1$

for $i \leftarrow 0$ **to** N **do**
  $interactionData[i] = envData[i] * snpData[i]$

Algorithm 7: Applying statistic model

Data: Two vectors of length N, $interactionData$ and $dataVector$
i is an index from 0 to $N - 1$

for $i \leftarrow 0$ **to** N **do**
  if $interactionData[i]$ ***not*** 0 **then**
    $dataVector[i] = 0$

3.2.5 Wrappers

CUDA and BLAS functions and interfaces are in C style. To make it easier to use and replace if needed they were put in wrappers. These wrappers are in C++ style and do the actual function calls to CUDA and BLAS. Since CUDA and BLAS contain a lot of various functions functions were added to the wrappers when needed.

There are several reasons to use wrappers. One is that some of the functions does not use exceptions but return an error code instead. This forces the program to check these error codes. This is hidden by using a wrapper that can throw the correct exception based on the error code. Exceptions are also better for performance because these remove the need for the *if* statements checking the returned error codes. The compiler can optimize better with checks for exceptions(e.g. try catch statements) because it can assume that the occurrence of exceptions is low. However usually checks for exceptions are not needed because they commonly signify an error that the program can not recover from.

Another reason is the lack of object orientation. For instance the BLAS interface standard is not object oriented which causes its functions to have a lot of parameters. Matrix vector multiplication with MKL(Intels version of BLAS) is shown in 5. The function call uses 12 parameters, most of which are of similar types. One time during the thesis it took a day to find an error that was in one of the CUBLAS calls. Due to a parameter being 1 instead of 0 it performed the operation $y = z * x + y$ instead of $y = z * x$. These types of errors are much easier to make when a function has a larger number of similar parameters. MKL is wrapped in **MKLWrapper**. Its wrapped version of matrix vector multiplication, shown in 6, has five parameters of which two have default values.

As is sometimes common BLAS function names are heavily abbreviated. For instance a function for matrix vector multiplication is **sgemv**, the s stands for single precision, ge for general, m for matrix and v for vector. General means that it is standard matrices and vectors, e.g. not triangular or sparse. The equivalent function in **MKLWrapper** can be found in example 6. The name matrixVectorMultiply instantly tells the user what it does. As mentioned earlier in section 3.2 the CuEira is either single or double precision depending on an option when compiling so **MKLWrapper** doesn't mention the precision. All matrices and vectors are also general. However even if the function name would contain that information too it would still not be so long that it was in the way. A possible name by using namespaces for the precision and type could be General::SinglePrecision::matrixVectorMultiply.

Example 5: Single precision matrix vector multiplication using MKL, $result = \alpha * matrix * vector + \beta * result$

Data: *layout*, storage type of the matrix, either column major(CblasColMajor) or row major(CblasRowMajor)
trans, use transpose, conjugate or neither on the matrix
matrix, *vector* and *result* are float pointers
α and β are constants
m, number of rows in the matrix
n, number of columns in the matrix
ld_matrix, leading dimension of the matrix
inc_vector and *inc_result* are the increments of the elements

cblas_sgemv(*layout*, *trans*, m, n, α, *matrix*, *ld_matrix*, *vector*, *inc_vector*, β, *result*, *inc_result*);

Example 6: Matrix vector multiplication using MKLWrapper, $result = \alpha * matrix * vector + \beta * result$

Data: *matrix*, HostMatrix
vector, HostVector
result, HostVector
α, constant, default 1
β, constant, default 0

matrixVectorMultiply(*matrix*, *vector*, *result*, α, β);

CUBLAS is wrapped together with the kernels made for CuEira in **KernelWrapper** in a similar way to how MKL is wrapped in **MKLWrapper**. **KernelWrapper** is explained in section 3.2.6. Other CUBLAS utility and CUDA functions are wrapped in **CudaAdapter**. The **CudaAdapter** has a functions to convert CUBLAS error codes to strings, allocate memory and so on.

The CUDA streams are managed by a **Stream** class and its factory. When the **StreamFactory** creates a new Stream object it creates a new CUDA stream, a new CUBLAS handle and associates the CUBLAS handle with the new stream. The **Stream** class has functions to retrieve them and if the Stream object is destroyed, it destroys both the CUDA stream and the CUBLAS handle. Each Stream object is associated with a **Device** object. The **Device** class represents a device(i.e. GPU) and since a thread can only issue commands to one device at a time the **Device** class has functions to set it as the active one and check if itself is the device active or not. The outcomes of the individuals does not change so it can be shared between all streams on the device and the **Device** class is responsible for the vector with the outcomes on the device.

The transfers to and from the device are done by two classes, **DeviceToHost** and **HostToDevice**. They perform asynchronous transfers on the corresponding matrix and vector class. It can either create a new container to transfer to or transfer to an given place in the memory. The former is useful for instance when transferring a group of vectors into a combined matrix.

3.2.6 Kernels

All the kernels are wrapped together with the CUBLAS kernels in **KernelWrapper**. All the kernels made for CuEira are in a similar style. They all do operations on each element of vectors. This makes their structure simple, each thread does the operations on one index. An example of one of the kernels and its wrapper function can be found in algorithm 8 and algorithm 9.

Algorithm 8: Kernel for vector addition

Data: *vector*1, *vector*2 and *result* are vectors as pointers,
length is the length of the vectors

```
__global__ void ElementWiseAddition(const PRECISION* vector1,
const PRECISION* vector2, PRECISION* result, const int length)
{
    int threadId = blockDim.x * blockIdx.x + threadIdx.x;

    if (threadId < length)
    {
        result[threadId] = vector1[threadId] + vector2[threadId];
    }
}
```

Algorithm 9: Wrapper for the kernel in algorithm 8

Data: *vector*1, *vector*2 and *result* are DeviceVectors of same length
cudaStream is the stream for the kernel
numberOfThreadsPerBlock is 256

```
void elementWiseAddition(const DeviceVector& vector1, const
DeviceVector& vector2, DeviceVector& result)
{
    //Error check for the length of the vectors in the source code removed

    const int numberOfBlocks = std::ceil(((double)
    vector1.getNumberOfRows()) / numberOfThreadsPerBlock);

    Kernel::ElementWiseAddition<<<<
    numberOfBlocks, numberOfThreadsPerBlock, 0, cudaStream >>>
    (vector1.getMemoryPointer(), vector2.getMemoryPointer(),
    result.getMemoryPointer(), vector1.getNumberOfRows());
}
```

3.2.7 Model

The calculations are done by using a group of **Model** classes. A list of these classes and their responsibilities is shown below.

Model
The model to calculate.

ModelConfiguration
Holds the data for the Model.

ModelResult
Contains the results from one Model on one set of data.

CombinedResults
The results for one set can consist of more than one ModelResult. CombinedResults have that responsibility.

ModelHandler
Has a **next** function to iterate over the data delegating it to DataHandler. It also has a pure virutal function **calculate** which should be used for calculating the Models and collecting the results.

ModelInformation
Contains the ContingencyTable and AlleleStatistics.

The point of these classes is that they provided common interfaces and are easy to extend. This makes it easy to later add or change which models are calculated and how their results are handled. The next section will explain how the Model base classes were extended to form the classes for the LR model.

3.2.7.1 Logistic Regression

The LR model is implemented using the **Model** base classes talked about in the previous section. There is a CPU version and a GPU version using CUDA. Some classes are shared between them because of common elements. The structure is shown in figure 3.1. A more detailed layout of the **ModelHandler** for **LogisticRegression** is shown in figure 3.2.

The LR algorithm is split into several protected functions so that each part can be tested individually. The parts are then used in the **calculate** function. For **CudaLogisticRegression** the parts corresponding to line 10 and 11 in algorithm 3 are done on the host. All the other parts of the LR algorithm are done by using CUBLAS or the kernels explained in section 3.2.6. The calculations for line 10 and 11 are in the **LogisticRegression** class so the code is shared between **CpuLogisticRegression** and **CudaLogisticRegression**.

Line 10 and 11 are done on the host instead of the device because the inverse of the information matrix(variable **J** in the algorithm) is done by SVD and no suitable GPU function for it was found. The reason for using SVD instead of normal matrix inverse was previously explained in section 2.6.1. CUBLAS does not provide an SVD function. However there are other libraries that do, CULA Tools [42] and MAGMA [43], however neither could be used.

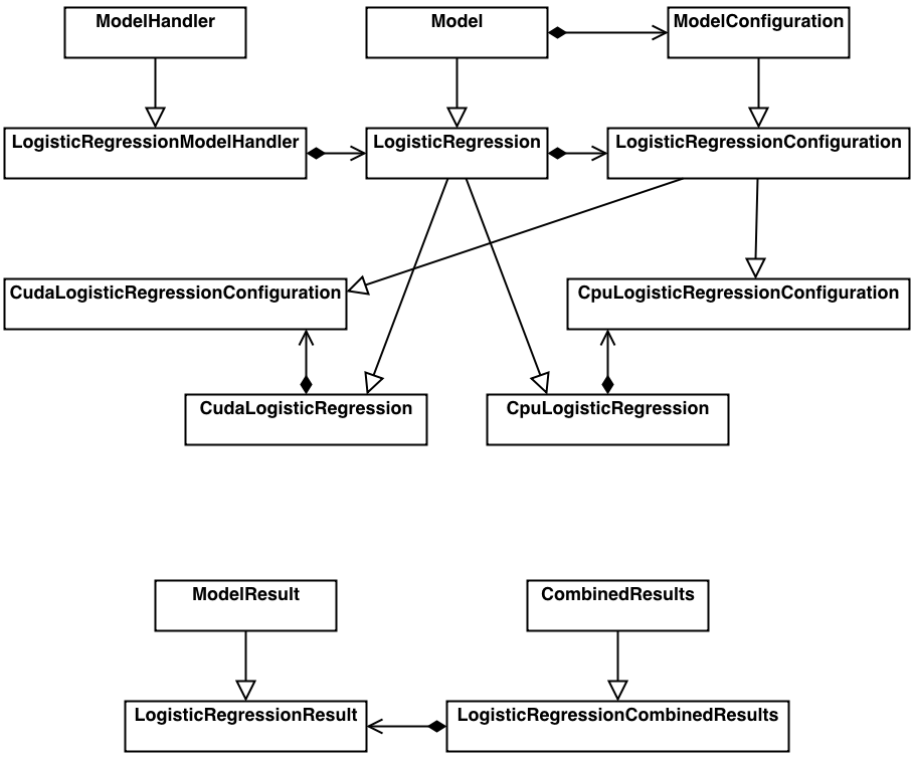


Figure 3.1: *Overview of the logistic regression classes. White arrow is inheritance. Diamond is that the class with the diamond has instances of the other class.*

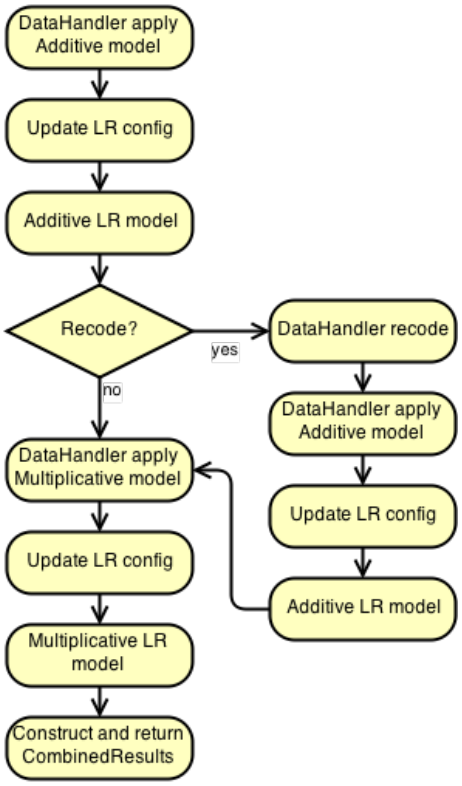


Figure 3.2: *Layout of the LogisticRegressionModelHandler*

CULA Tools could not be used because it uses streams internally in the functions so it can not be executed on streams [69]. It also uses device synchronisation which means that the all streams will be synchronised every time the function is used [69]. MAGMA can use streams for its BLAS functions, however it does not for the other functions including the SVD [70].

The size of the vectors and matrices used in that part are also small, their size is 4 + the number of covariates. However it can sometimes be better to calculate even small matrices and vectors on the device to avoid the transfers that would be needed otherwise [34]. An suitable GPU SVD kernel might be faster.

After transfers it is necessary to wait for the transfer to finish to prevent errors with using the data before it has been transferred completely. This is done by syncing the thread with the stream, it forces the thread to wait until all current kernels and transfers on the stream is complete. However due to the problem with asynchronous transfers and older architectures described in section 2.3.2 there is also an option when compiling to synchronise after each kernel.

3.2.8 Worker Thread

A number of worker threads use the previously explained classes to perform the work. One worker thread corresponds to one stream on a GPU. Normally three streams are used per GPU but it can easily be changed by changing a parameter in the main **Configuration** class. The layout of the worker thread is shown in figure 3.3.

The thread has its own **DataHandler**, **ModelHandler**, **ModelConfiguration** and Model. Some objects are shared with the other workers, most notably the **ResultFileWriter** and the **DataQueue**. Each thread loops over the **ModelHandlers next** function, tells the **ModelHandler** to calculate and then sends the results to the **ResultFileWriter**.

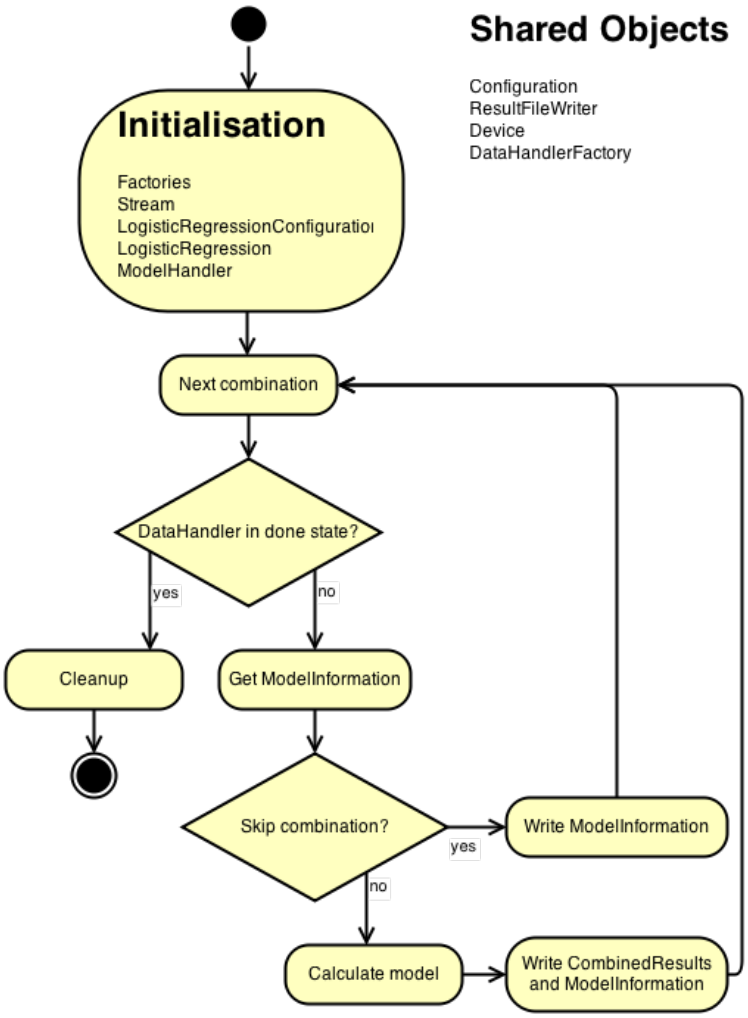


Figure 3.3: *Structure of the worker thread*

3.2.9 Testing

Almost all classes have a corresponding unit test that tests its functionality in isolation. However the matrix and vector classes are never mocked out because they are too basic and fundamental. It does mean that if there are multiple errors when testing and the tests for the matrix and vectors failed the fault likely is with those. However the number of integration tests are few and there is a need to create more.

Since mocks are created by using inheritance from the class to be mocked all functions need to be virtual. Some classes also have empty protected constructors to be used by the mock, others have helper functions to construct the mock. This goes against some parts of clean code but is necessary because of how C++ and mocking works. It could potentially be improved by using private constructors and friends to give the mock access to the private constructor.

The data for the tests where calculated by hand or by using Matlab.

3.2.10 Memory Usage

The memory usage is low because only the SNPs whose models currently is being calculated is stored in memory. This also means that most of the memory usage depends on the number of individuals not the number of SNPs. The memory is also bound by the GPU rather than host memory because of the relative larger size of the host memory. The amount used depends on the precision, single precision floating point numbers are four bytes while double precision numbers are eight bytes.

3.2.10.1 Device Memory

The variables used on the device and their number of elements are shown in table 3.1. The memory need with T number of streams is shown in equation 3.1.The amount of individuals than can fit inside 6GB(i.e. a typical GPU) with three streams for different number of covariates and precision is shown in table 3.2. The number of individuals possible is one magnitude or more than the an usual study.

$$(N + T(2NM + 2N + 2M + MM)) * ElementSize \tag{3.1}$$

Matrix/vector	Size	Instances
Outcomes	N	1
Predictors	$N \times M$	One per stream
Probabilities	N	One per stream
Scores	M	One per stream
Beta	M	One per stream
Work Area	N	One per stream
Work Area	$N \times M$	One per stream
Information Matrix	$M \times M$	One per stream

Table 3.1: *Sizes of the variables stored on the GPU, N = number of individuals, $M = 4+$ number of covariates*

Number of covariates	Single precision	Double precision
0	48.4	24.2
5	24.6	12.3
10	16.5	8.2
20	9.0	5.0
100	2.4	1.2

Table 3.2: *The number of individuals in millions needed to fill 6GB with the given precision, number of covariates and three streams*

3.2.10.2 Host Memory

The matrix and vector variables are shown in table 3.3. However there are also other data related information that needs to be stored and that potentially uses a significant amount of memory, they are shown in table 3.4. Classes that only exist in few instances and similar are ignored because their effect on the memory is minimal compared to the actual data. With the sizes from the tables the needed amount of host memory is shown in 3.2.

Equation 3.2 shows how much memory is used with N individuals, E environmental factors, S SNPs, C covariates and T streams. str is the size of a string, it can vary however due to overhead they can reach up to 100 bytes. Assuming 100 bytes for each string the memory requirement is low. With extreme values of 10 million SNPs, 10 million individuals, 1000 covariates and double precision it needs 88 GB, which is large, but not impossible to fit inside the memory of a modern cluster.

$$ElementSize*(N(1+E+C)+T(60+6N+31C+C^2))+str*(N+E+3*S)+12N+4E+28S \tag{3.2}$$

Matrix/vector	Size	Instances
Outcomes	N	1
Environment data	$N \times E$	1
Covariates data	$N \times C$	1
SNPVector	$2N$	One per stream
EnvironmentVector	$N \times (4 + C)$	One per stream
InteractionVector	$N \times (4 + C)$	One per stream
Beta	$(4 + C)$	One per stream
Scores	$(4 + C)$	One per stream
Sigma	$(4 + C)$	One per stream
uSVD	$(4 + C) \times (4 + C)$	One per stream
vSVD	$(4 + C) \times (4 + C)$	One per stream
Work area	$(4 + C) \times (4 + C)$	One per stream

Table 3.3: *Sizes of the matrix/vector variables stored on the host*

Object	Contains	Number
SNP	3 strings, 6 int, 1 bool	S
EnvironmentFactor	1 string, 1 int	E
Persons	1 string, 2 int, 1 bool	N

Table 3.4: *Sizes of the other variables stored on the host*

Chapter 4

Results

This chapter contains the results of profiling the program. The data used consists of randomized data with equal number of cases and controls. The number of SNPs, individuals and covariates was varied. The specifications for the hardware used is shown in table 4.1. The GPUs are of the Fermi architecture so they have one queue for the kernels and transfers.

The expectations were that CuEira would have linear efficiency with multiple GPUs since there is no communication between the GPUs and that each gene-environment combination is calculated independently.

CPU	2 Intel Xeon E5620, 4 cores per CPU
GPU	4 Tesla C2050
Memory	48 GB ram

Table 4.1: *Hardware specifications*

All the execution times used here are the times the calculation part of the program took. This is because the initialisation part of the program varies a lot in time. The part of the initialisation that varies greatly is when creating the vector of the outcomes(i.e. the individuals phenotypes) which. The total time for initialisation and the time to create the outcomes are shown in figure 4.1. Except for the first execution the order does not seem to matter, figure 4.2 shows the initialisation times for the same executions as figure 4.1 however they were performed in opposite order. To prevent a bias from this in the results all the plots use the calculation time instead of total run time.

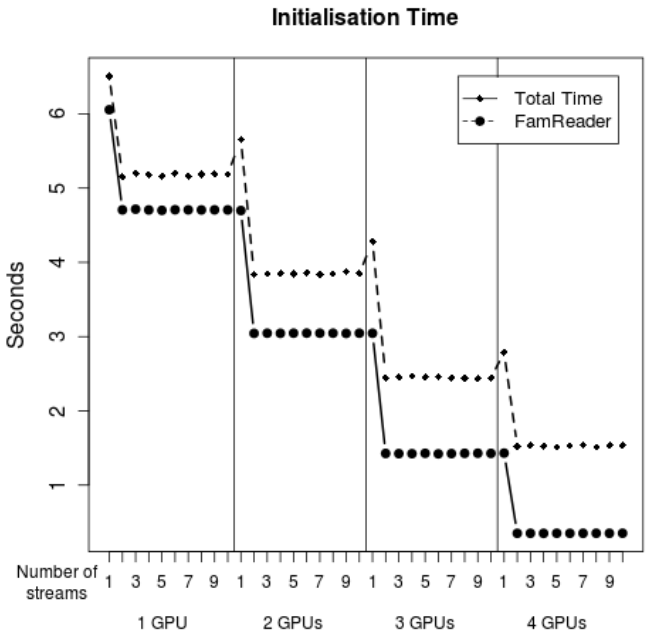


Figure 4.1: *10 000 SNPs and individuals with 1-4 GPUs with 1-10 streams. It was performed starting with 1 GPU and 1 stream and then increasing the streams and then the GPUs. I.e. from left to right in the figure.*

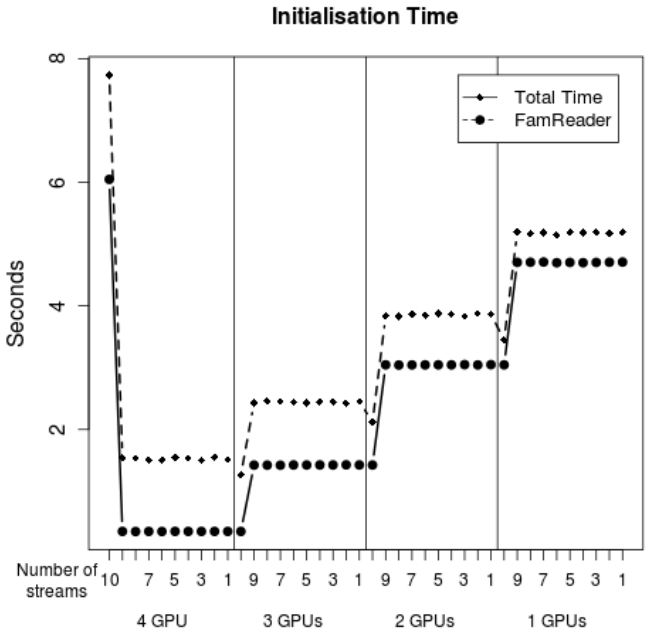


Figure 4.2: *Same as figure 4.1 but calculating the files in opposite order.*

4.1 Scaling

This section contains the results of how the programs speed changes when the size of the data is varied.

The program is expected to be $O(n)$ for the SNPs because the program treats each SNP-environment combination independently, more SNPs simply means doing the same things more times. Figure 4.5 and 4.6 shows the calculation time for various number of SNPs relative to the time with 10 000 SNPs according to equation 4.1. When it is one then the increased time is what was expected. As the figures show all points are close to one except one point. Also due to how long time the calculations take with a large number of SNPs it is also not possible to perform all profiling with large number of SNPs.

$$\frac{time_{X\ SNPs}}{time_{10^4\ SNPs}} / \frac{X}{10^4} \tag{4.1}$$

Individuals increase work but it does not increase so that double means double the work, shown in figure 4.7. There are no major changes with the addition of covariates. With 20 covariates is shown in figure 4.8. Individuals are also a part of the LR algorithm so might affect things that way.

The number of covariates affects the speed when the number of individuals is low, see figure 4.9. However when the number of individuals is large the effect of the covariates is much smaller as shown in figure 4.10.

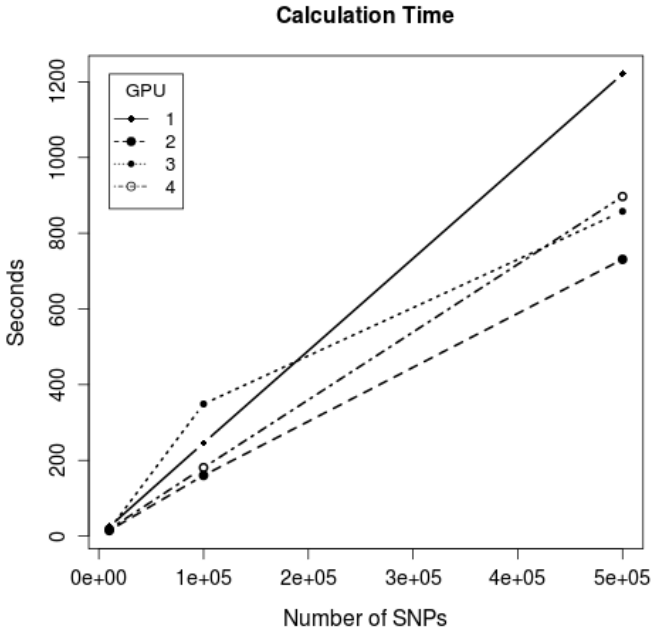


Figure 4.3: *Varying the number of SNPs. 10 000 individuals, 0 covariates, 4 streams*

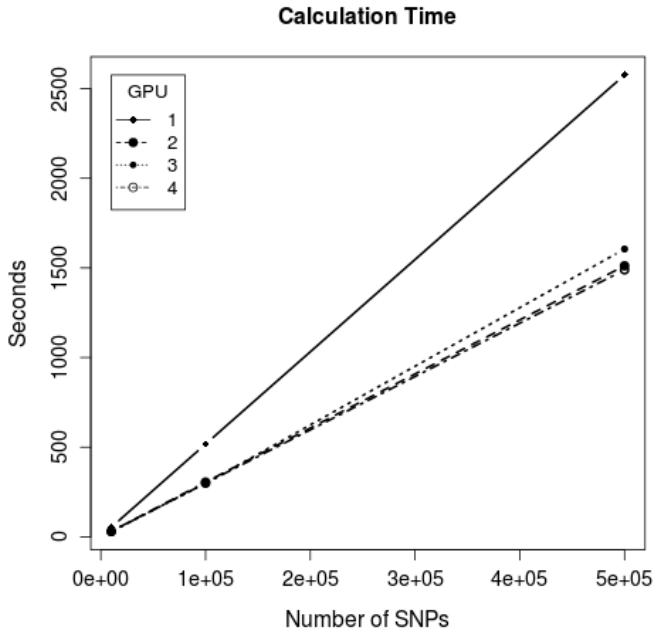


Figure 4.4: *Varying the number of SNPs. 10 000 individuals, 20 covariates, 4 streams*

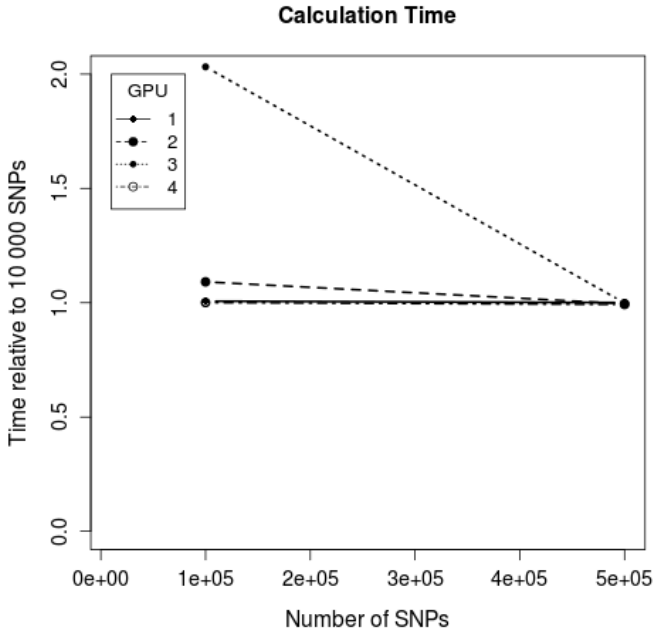


Figure 4.5: *Relative time versus 10 000 SNPs. 10 000 individuals, 0 covariates, 4 streams*

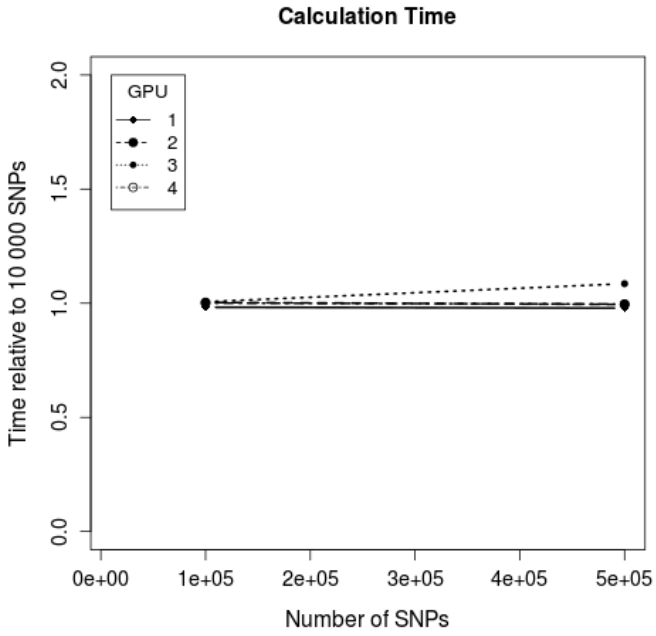


Figure 4.6: *Relative time versus 10 000 SNPs. 10 000 individuals, 20 covariates, 4 streams*

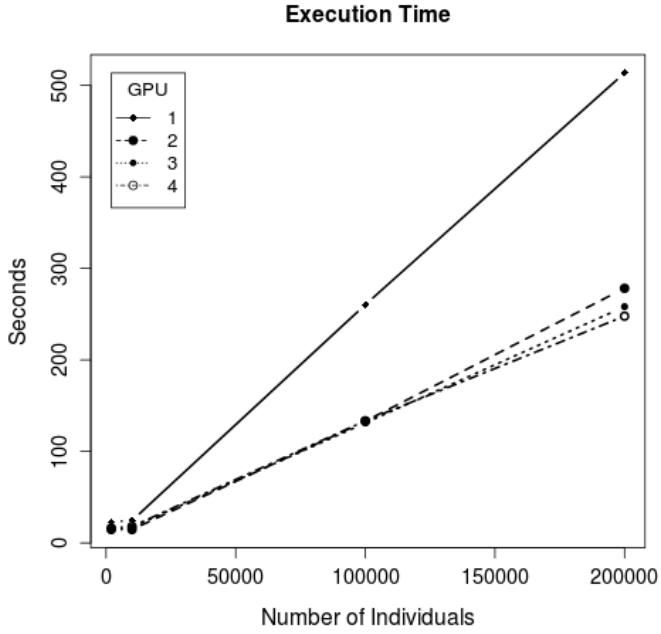


Figure 4.7: *Varying the number of individuals. 10 000 SNPs, 0 covariates, 4 streams*

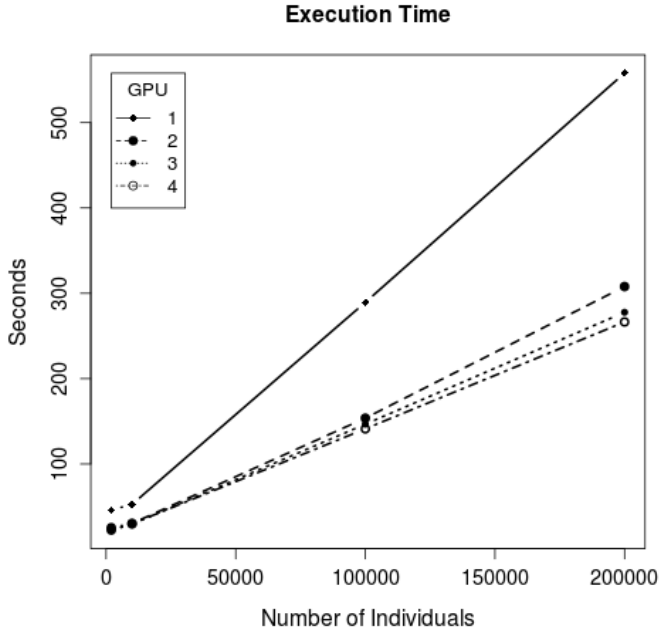


Figure 4.8: *Varying the number of individuals. 10 000 SNPs, 20 covariates 4 streams*

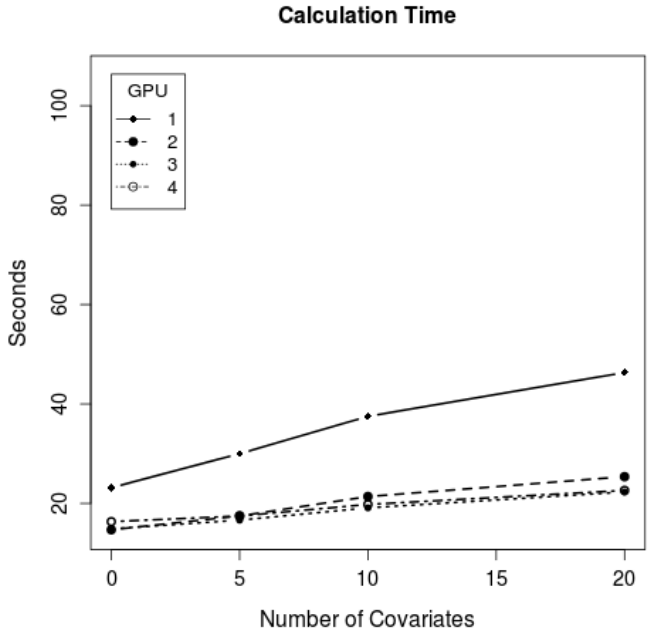


Figure 4.9: *Varying the number of covariates. 10 000 SNPs, 2 000 individuals*

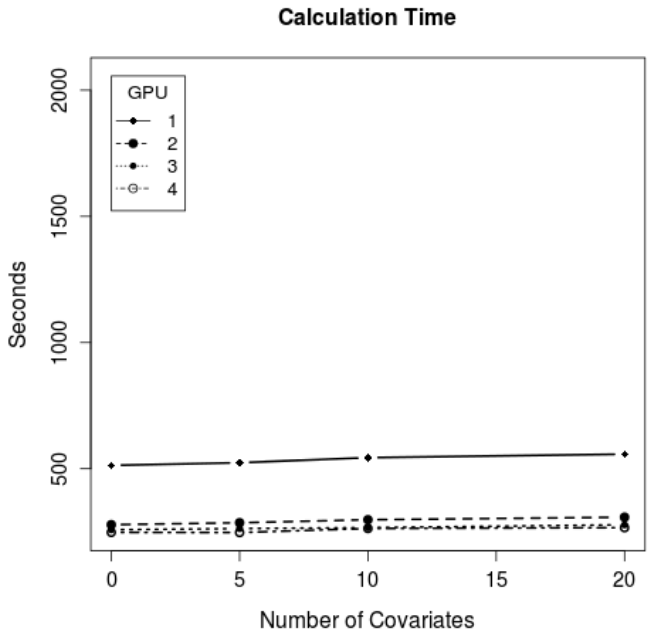


Figure 4.10: *Varying the number of covariates. 10 000 SNPs, 200 000 individuals*

4.2 Streams and Multi-GPU

The number of streams affects the performance significantly. As figure 4.11 and figure 4.12 show using too few streams increases the time the program takes. This is because to few streams does not provide enough work for the GPU. The best performance is when the total number of streams is slightly above the number of cores. It likely means that the CPU is the bottleneck. However for some data sizes with one GPU the GPU is the bottleneck as figure 4.11 shows where increased number of streams beyond six gives little. Using too many streams have some negative effects.

NVIDIAs profiler nsight shows warning messages about low overlap between transfers and kernels and between kernels and kernels when using four GPUs and four streams. This also points at the CPU can not provided enough data to the GPUs. However these warnings does not show up for two GPUs and four streams.

Figures 4.14, 4.15, 4.16, 4.17, 4.18 and 4.19 show how the program behaves and scales with multiple GPUs. The figures 4.14 and 4.15 shows the calculation time in seconds. Figures 4.16 and 4.17 shows the speed up for figures 4.14 and 4.15, while figures 4.18 and 4.19 show their efficiency.

Two GPUs show increased speed compared to one GPU. However for three and four GPUs the speed is almost the same as with two GPUs. With low number of individuals(2000) the scaling to two GPUs almost reaches linear efficiency, 10 000 individuals also has decent scaling to two GPUs. Three and four GPUs show almost no increase in speed for any amount of data therefore also shows low efficiency, in some cases there even is a drop in speed.

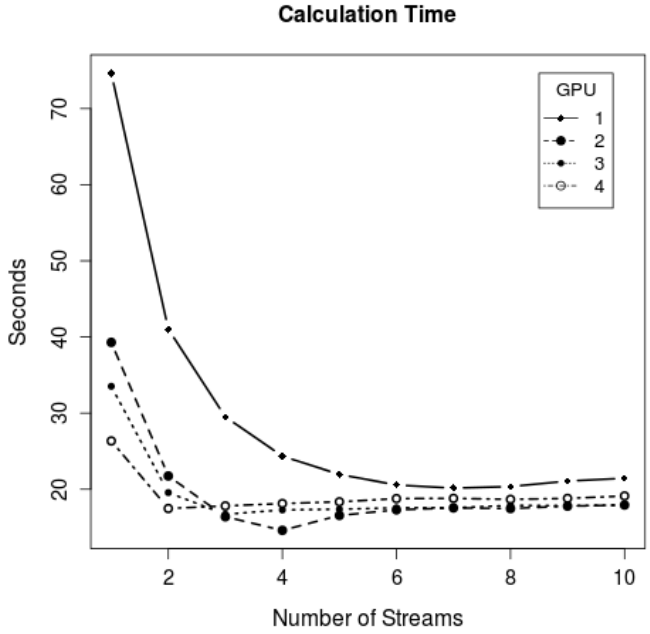


Figure 4.11: *1 to 10 streams, 10 000 SNPs, 10 000 individuals*

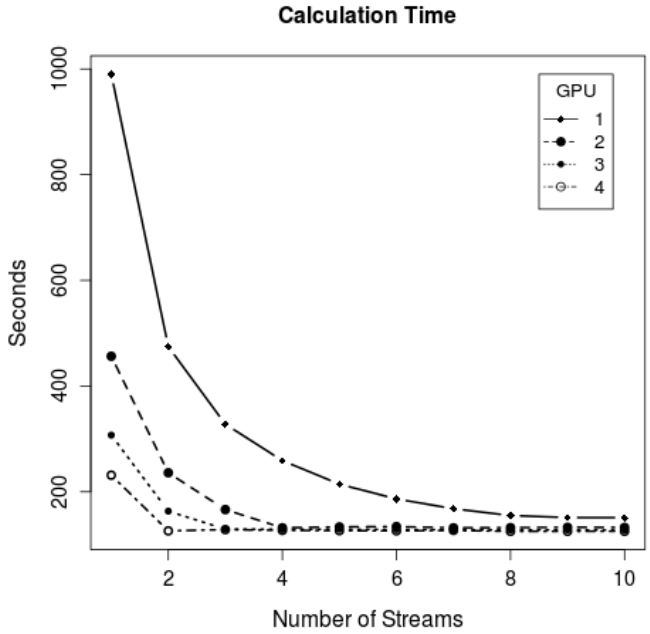


Figure 4.12: *1 to 10 streams, 10 000 SNPs, 100 000 individuals*

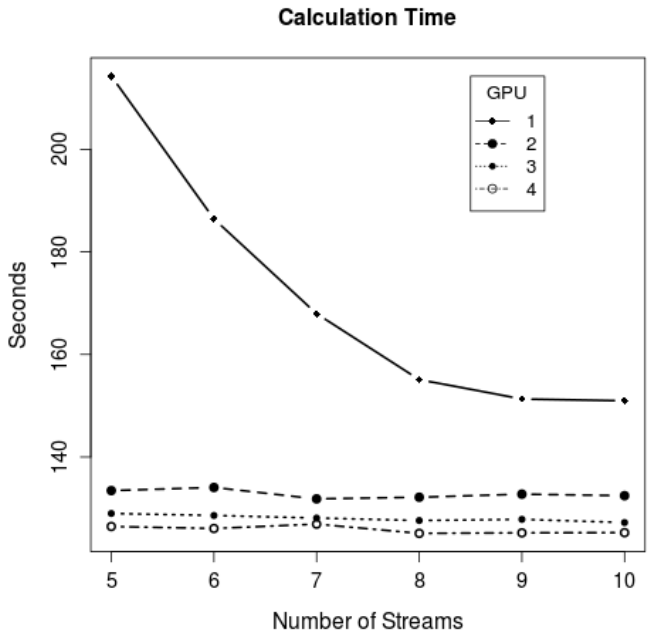


Figure 4.13: *5 to 10 streams, 10 000 SNPs, 100 000 individuals*

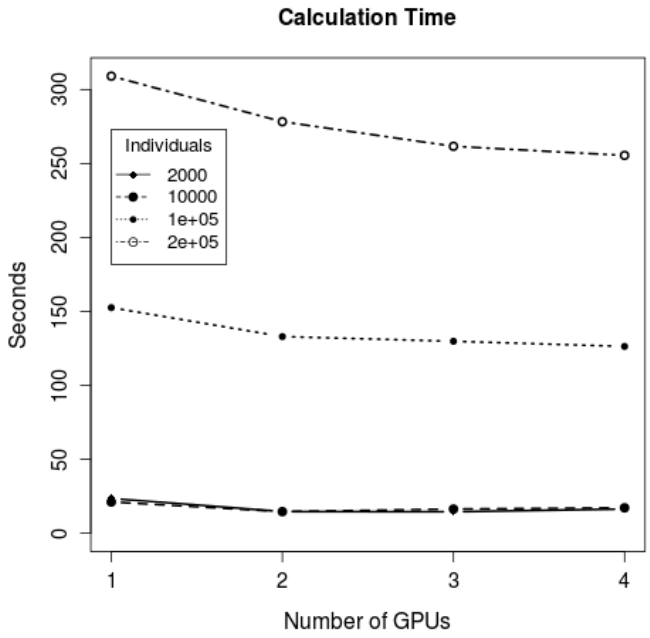


Figure 4.14: *Varying the number of GPUs and individuals. Number of streams are what give best performance. 10 000 SNPs, 0 covariates*

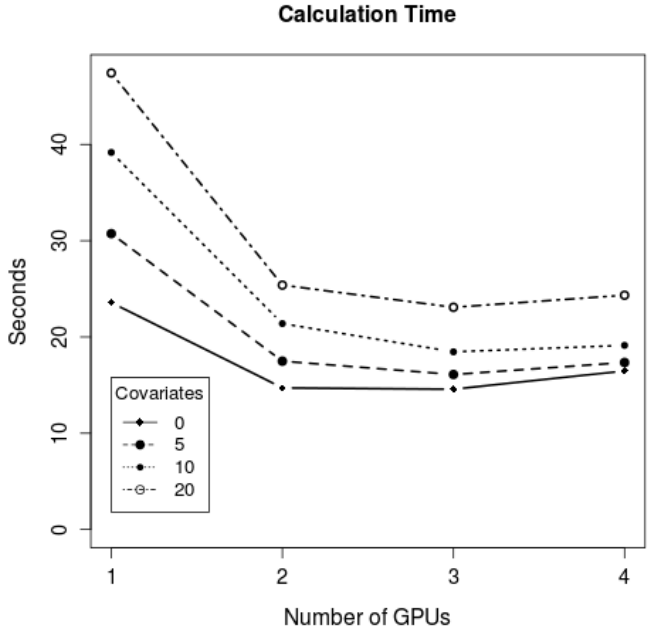


Figure 4.15: *Varying the number of GPUs and covariates. Number of streams are what give best performance, 10 000 SNPs, 2 000 individuals*

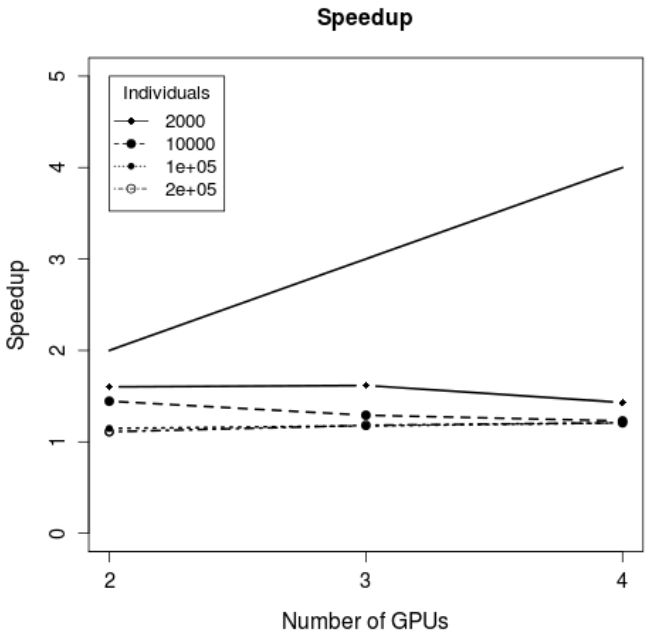


Figure 4.16: *Speedup vs one GPU for figure 4.14. Plain diagonal line corresponds to linear efficiency*

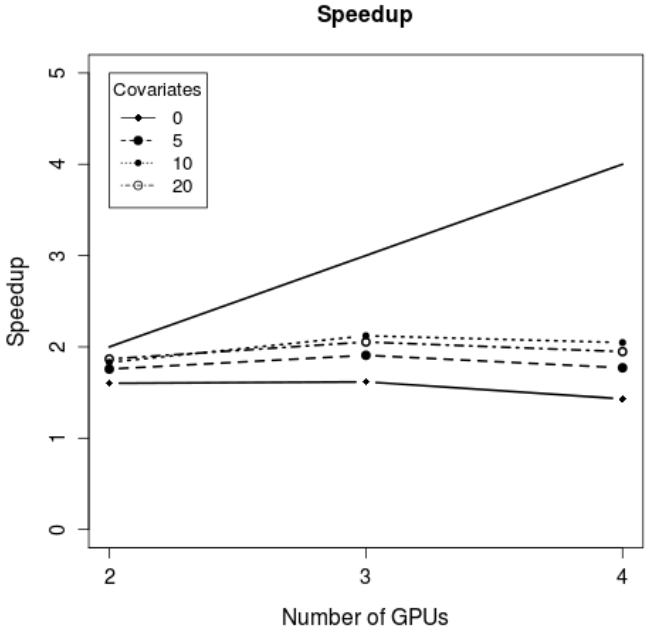


Figure 4.17: *Speedup vs one GPU for figure 4.15. Plain diagonal line corresponds to linear efficiency*

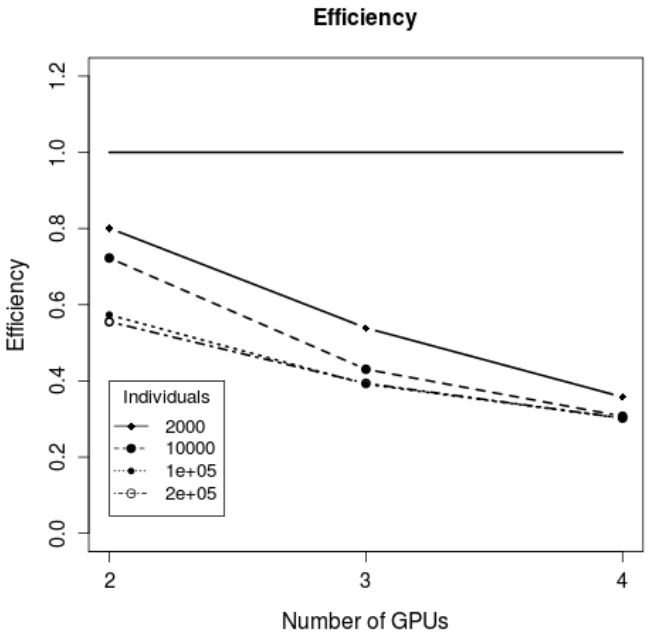


Figure 4.18: *Efficiency vs one GPU for figure 4.14. Plain horizontal line corresponds to linear efficiency*

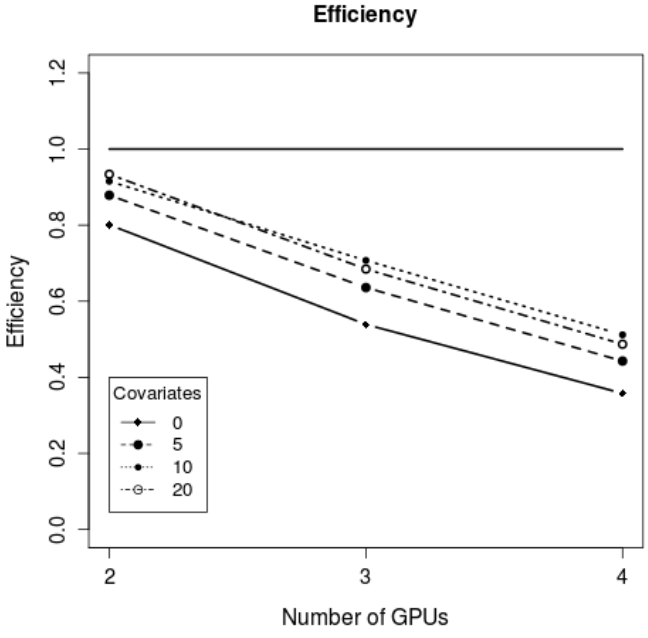


Figure 4.19: *Efficiency vs one GPU for figure 4.15. Plain horizontal line corresponds to linear efficiency*

4.3 Single versus Double Precision

Using double precision increases the time in almost all cases. Figure 4.20 and 4.21 show the relative difference in calculation time for single precision divided by double precision with the optimal number of streams. One point has better performance with double precision than single.

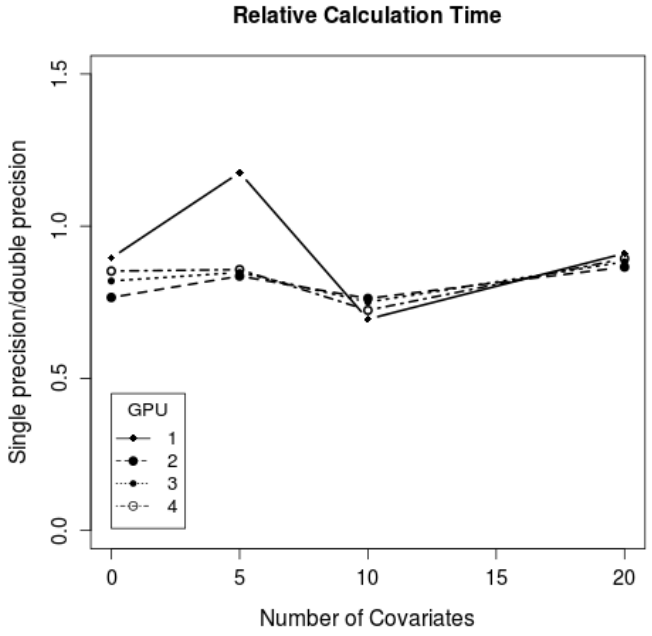


Figure 4.20: *Relative calculation time for single precision divided by the corresponding time for double precision. Number of streams are what give best performance, 10 000 SNPs, 10 000 individuals*

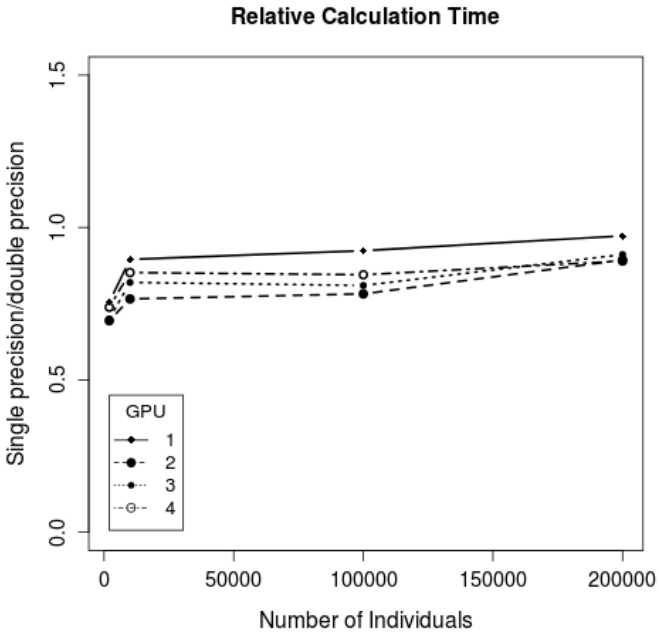


Figure 4.21: *Relative calculation time for single precision divided by the corresponding time for double precision. Number of streams are what give best performance, 10 000 SNPs, 0 covariates*

4.4 The LR algorithm, Transfers and Synchronisations

With GPUs Fermi architecture a problem can occur when using asynchronous transfers as talked about in section 2.3.2. To examine if it affects the speed of the program several more synchronisations were added so that the CPU synchronises after each kernel or transfer call instead of when only necessary. A comparison between the speed with and without these is shown in figures 4.23 and 4.22. More synchronisation increases the time the calculations took in most cases. However with sufficient number of individuals(100 000 and up) some speed is gained with more synchronisations. 200 000 individuals have increased speed in most cases. The transfer times relative to the calculations in LR decreased with the increased syncing, see figure 4.24 and 4.26.

As figure 4.28 shows with increased number of covariates the CPU time of the LR algorithm increases. However with more individuals the effect of the covariates decreases see figure 4.29. When the number of GPUs increase the time spent on transfers increases, compare figure 4.24 with figure 4.25.

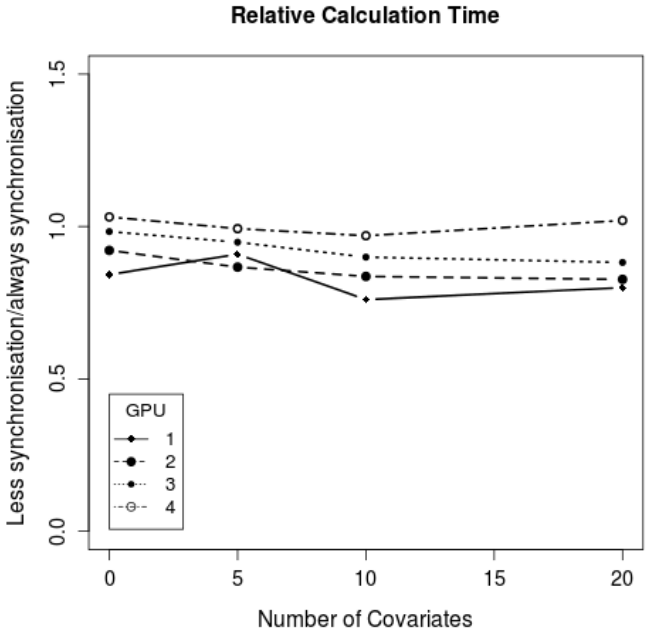


Figure 4.22: *The calculation time without the increased synchronisations divided by the time with increased synchronisations. 10 000 SNPs, 2 000 individuals*

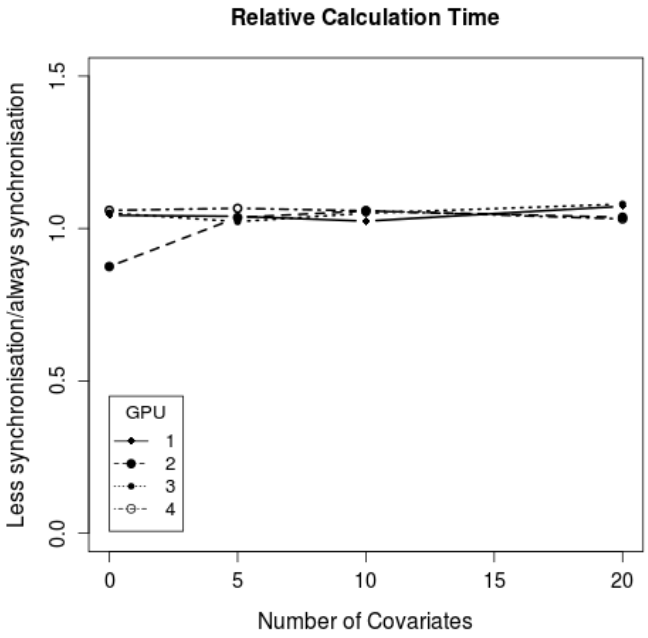


Figure 4.23: *The calculation time without the increased synchronisations divided by the time with increased synchronisations. 10 000 SNPs, 200 000 individuals*

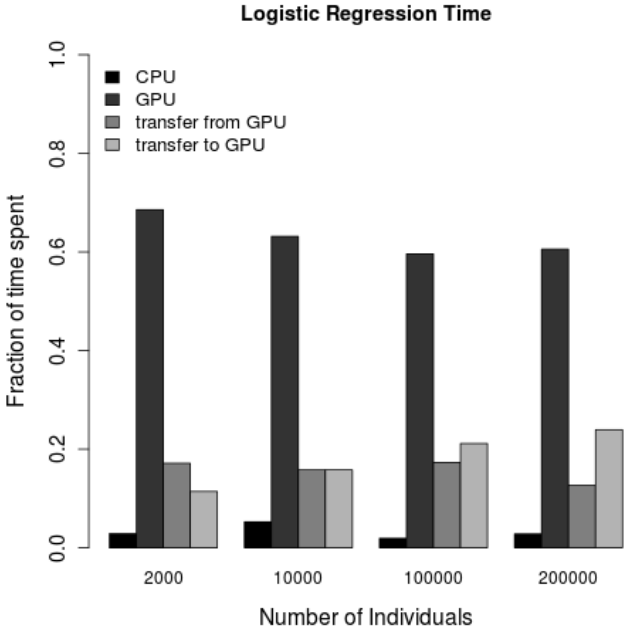


Figure 4.24: *Distribution of time spent on the LR algorithm. 1 GPU, 3 streams, 10 000 SNPs, 0 covariates*

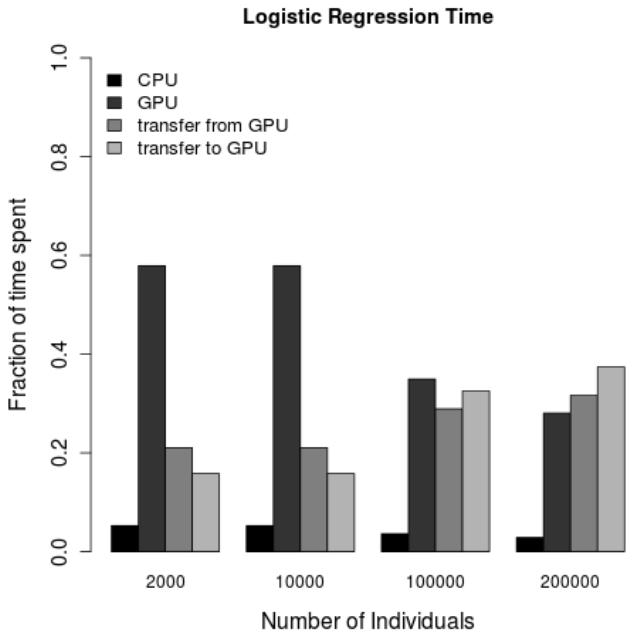


Figure 4.25: *Distribution of time spent on the LR algorithm. 4 GPU, 3 streams, 10 000 SNPs, 0 covariates*

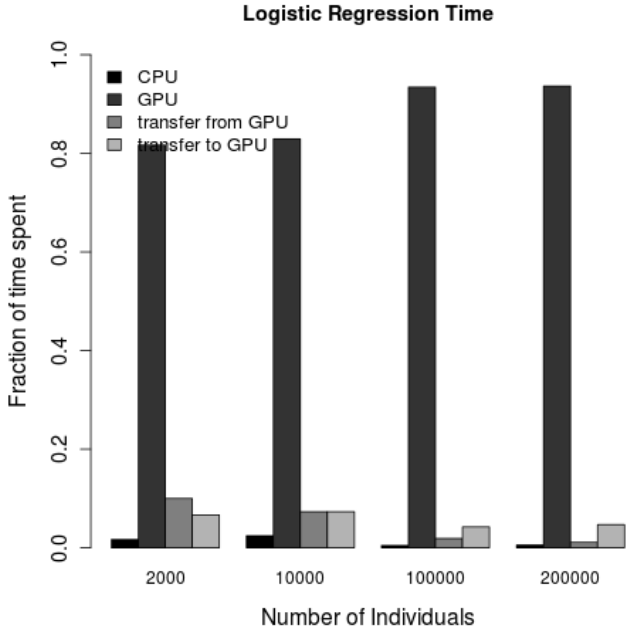


Figure 4.26: *Distribution of time spent on the LR algorithm with increased synchronisations. 1 GPU, 3 streams, 10 000 SNPs, 0 covariates*

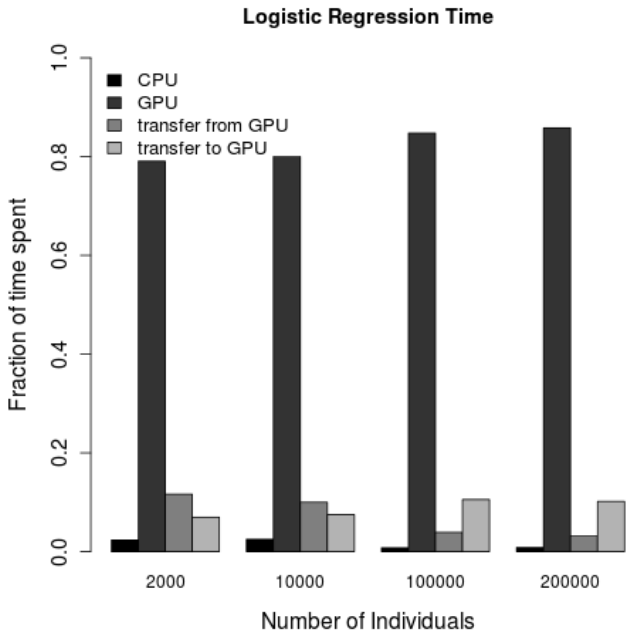


Figure 4.27: *Distribution of time spent on the LR algorithm with increased synchronisations. 4 GPU, 3 streams, 10 000 SNPs, 0 covariates*

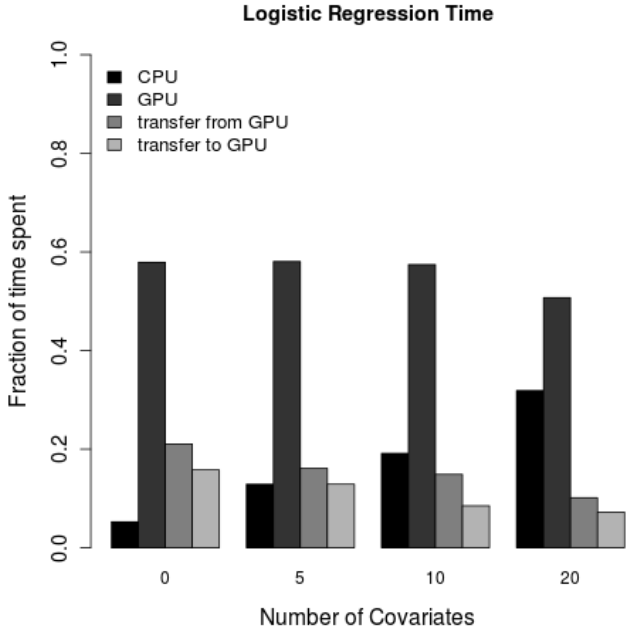


Figure 4.28: *Distribution of time spent on the LR algorithm. 4 GPU 3 stream, 10 00 SNPs, 2 000 individuals*

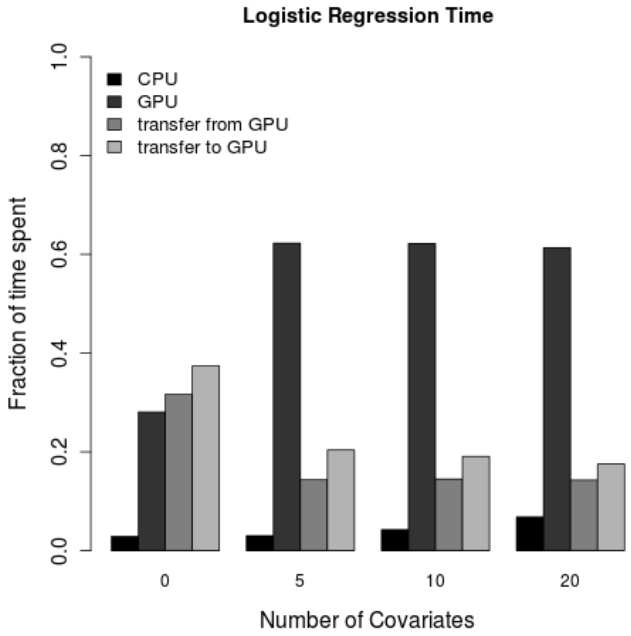


Figure 4.29: *Distribution of time spent on the LR algorithm. 4 GPU 3 stream, 10 000 SNPs, 200 00 individuals*

4.4.1 Time Distribution for Kernels

Table 4.2 shows how the time for doing calculations on the GPU is used based on the output from Nsight. Most of the time is spent on gemv2T which is matrix multiplication of a transposed matrix and a vector. It is used in the calculation of the scores vector, line 8 in algorithm 3. The custom kernels takes up a small part of the calculations unless the number of covariates is high, see table 4.3. About 10% of the time is spent on the element wise multiplication.

Kernel	Kernel type	10ki	100ki
gemv2T	CUBLAS	66.6	72.2
block dot	CUBLAS	18.9	14.4
gemv2N	CUBLAS	7.1	8.1
ElementWiseMultiplication	Custom	2.9	2.3
LogisticTransform	Custom	1.1	0.9
LogLikelihoodParts	Custom	0.8	0.6
ElementWiseDifference	Custom	0.7	0.6
VectorMultiply1MinusVector	Custom	0.7	0.5
dot kernel	CUBLAS	0.7	0.3
reduce	CUBLAS	0.5	0.1

Table 4.2: *Distribution of the time spent on kernels in the GPU calculations, numbers are in percentage of total time. 1 GPU, 9 streams, 0 covariates*

Kernel	Kernel type	10ki	100ki
gemv2T	CUBLAS	43	53.9
gemm	CUBLAS	36.1	25.8
ElementWiseMultiplication	Custom	11.8	10.6
gemv2N	CUBLAS	5.9	7.4
LogisticTransform	Custom	0.7	0.7
ElementWiseDifference	Custom	0.5	0.4
VectorMultiply1MinusVector	Custom	0.4	0.4
scal	CUBLAS	0.4	0.1
memset	CUBLAS	0.3	0.1
LogLikelihoodParts	Custom	0.6	0.5
dot kernel	CUBLAS	0.3	0.2
reduce	CUBLAS	0.2	0.1

Table 4.3: *Distribution of the time spent on kernels in the GPU calculations, numbers are in percentage of total time. 1 GPU, 9 streams, 20 covariates*

4.5 DataHandler

The **ResultFileWriter** and **DataQueue** each has a lock to prevent race conditions. They also have a timer for how much time that has been spent waiting at the locks. There was very little time spent at them, no more than a couple of seconds even for the larger files which took 10-30 mins. The distribution of time spent between reading the SNPs data, recoding and applying the statistical model is mostly the same for all data sizes and number of GPUs. Most of the time is spent reading the SNPs, above 80%. An example of some times is shown in figure 4.30. It is hard to say if it is a problem due to the parallel nature of the program. It is possible that the time to read the SNPs can be hidden by the GPU calculations.

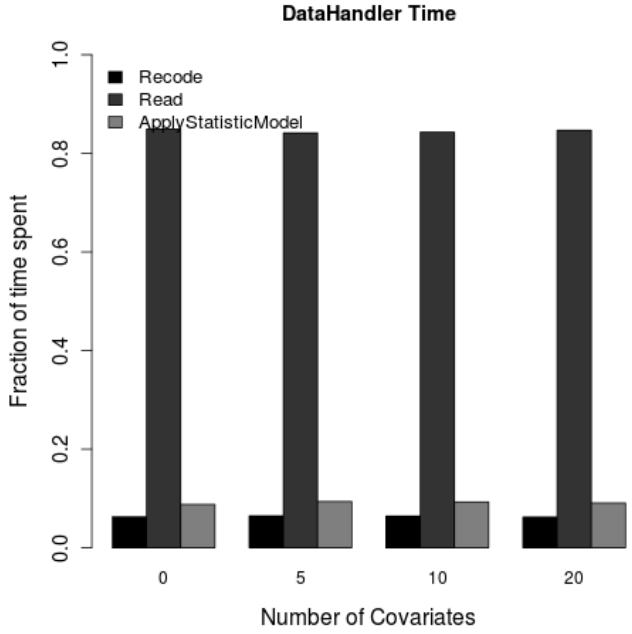


Figure 4.30: *Time distribution of reading SNPs, recoding and applying statistical model. 4 GPU, 3 streams, 10 000 SNPs, 200 000 individuals*

4.6 Comparison with GEISA

Since CuEira uses the same CPU with the addition of GPUs it is expected that CuEira will have higher speed. The speedup varies depending on the size of the data, higher number of individuals and covariates increases the speedup. GEISA had problems with the dataset with 200 000 individuals, it took much longer than expected and was cancelled before completing. It could be a memory problem since GEISA stores everything in memory. A different computer with more memory(96GB) did not have the same problem.

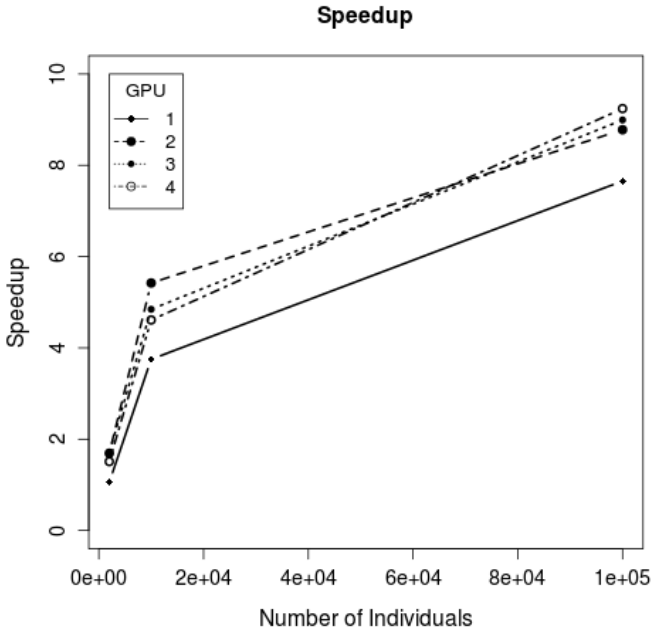


Figure 4.31: *Speedup GEISA vs CuEira. 10 000 SNPs, 0 covariates. 200 000 individuals were excluded because GEISA had problems with that amount of data.*

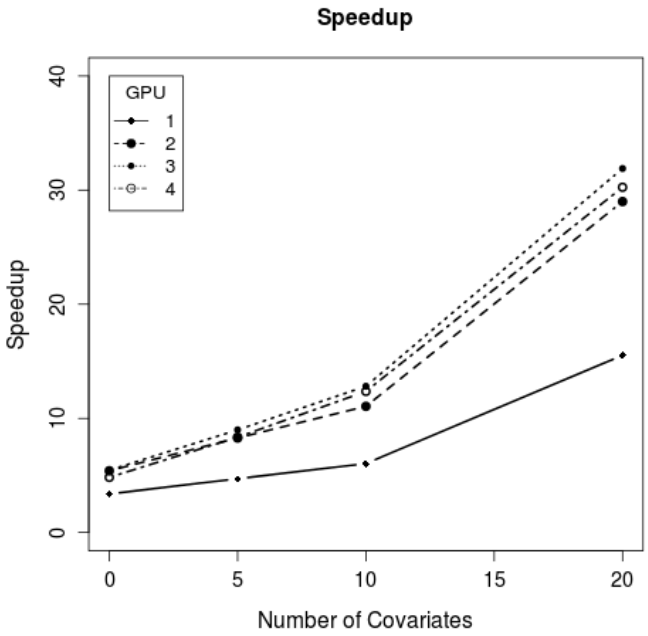


Figure 4.32: *Speedup GEISA vs CuEira. 10 000 SNPs, 10 000 individuals*

Chapter 5

Discussion and Conclusions

The CuEira implementation shows promising results, compared with GEISA it shows a significant speedup. CuEira also has almost full unit test coverage and is easy to extend and modify due to the modularisation. This will allow the program to have a longer lifetime. For instance the logistic regression could be switched for a different method without rewriting the other parts of the program.

CUBLAS was not much different to use than regular BLAS, however due to the higher number of parameters in their functions and lack of object orientation neither is user-friendly. Wrappers can make it easier to use. If the algorithm uses larger amounts of data than what fits inside the GPUs memory using GPUs are more complex to use than CPUs because of the more complex communications and transfers needed. Complex kernels can also be hard and time consuming to write and optimize. The wrappers made for CuEira could be used in the future for other projects and also be released as an library. The wrappers makes it easier to use basic CUDA and CUBLAS features.

As previously mentioned in earlier chapters CuEira has several compiling options that changes some of its properties. The results show that single precision has better performance than with double performance. Increased synchronisation gives better performance when the number of individuals is 100 000 and up. Increased synchronisation does always decrease the transfer times but it does not always result in better performance.

The programs efficiency with multiple GPUs could potentially be improved by moving more parts of the program to the GPU. This might reduce the amount of work the CPUs have to perform and improve the efficiency. If there is CPU power left it could be used for threads that perform the algorithm using CPU only. This would also lead to a version of the program that does not need GPUs. Optimizing the custom kernels made for the program could improve the performance for datasets with high number of covariates. However it should be done after more parts are moved to the GPU since that might change which kernels needs to be optimized.

Another potential improvement is to move the reading of the SNPs data from the file to separate threads and add a another queue. The new threads would

then take SNPs from the old queue, read the SNPs data and put it in the new queue. Then when a worker thread fetches a SNP from the new queue its data is already in the memory. The time needed to read the SNPs would be the same, however it would reduce the time a thread spends between finishing one combination and starting the next combination. If. However it might be the case that the GPU already has enough work from the other streams to cover the time spent reading the SNPs data.

Considering the efficiency of CuEira with several GPUs it is best to use it with one GPU, two if the number of individuals is low. However it depends on the GPUs and CPUs relative performance. Better CPU performance relative to the GPU might get scaling to multiple GPUs. The optimal number of streams per GPU is so that the total number of streams is the same or slightly higher than the total number of CPU cores.

Chapter 6

Outlook

GPUs are suitable for interaction depending on the method. They suit well with methods that consider each pair independently and that have extreme parallelism in the calculations for the combination. However doing even small parts on the CPU can reduce the performance. CUDA and CUBLAS could be easier to use after improvements to their interfaces or by using wrappers around them.

The next step to improve the speed of CuEira would be to move more parts of the algorithm to the GPU and by using clusters. It is likely that the program would scale well to clusters because of the embarrassingly parallel nature, which SNPs to calculate is the only communication necessary. One possible reason to why CuEiras efficiency for multiple GPUs is low is that the CPU can not provide enough computations to the GPUs. A method for validating the results also needs to be implemented, for instance bootstrap or permutation tests.

There is a need for better methods and definitions for interaction for non binary environmental factors. For interaction with binary environmental factors further speed might be gained by using gene-gene interaction methods since these methods often take advantage of the properties of binary variables. The definition of interaction could potentially be extended to equation 6.1 to cover non-binary environmental factors. x is the value of the environmental factor. The presence of interaction would then depend on the value of the environment factor.

$$OR^{x}_{interaction} > OR_{SNP} + OR^{x}_{environment} - 1 \tag{6.1}$$

Bibliography

[1] H. J. Cordell, "Detecting gene–gene interactions that underlie human diseases," *Nature Reviews Genetics*, vol. 10, no. 6, pp. 392–404, 2009.

[2] S. J. Winham and J. M. Biernacka, "Gene–environment interactions in genome-wide association studies: current approaches and new directions," *Journal of Child Psychology and Psychiatry*, vol. 54, no. 10, pp. 1120–1134, 2013.

[3] B. Ding, H. Källberg, L. Klareskog, L. Padyukov, and L. Alfredsson, "Geira: gene-environment and gene-gene interaction research application," *European Journal of Epidemiology*, vol. 26, no. 7, pp. 557–561, 2011.

[4] H. Mahdi, B. A. Fisher, H. Källberg, D. Plant, V. Malmström, J. Rönnelid, P. Charles, B. Ding, L. Alfredsson, L. Padyukov, *et al.*, "Specific interaction between genotype, smoking and autoimmunity to citrullinated α-enolase in the etiology of rheumatoid arthritis," *Nature genetics*, vol. 41, no. 12, pp. 1319–1324, 2009.

[5] K. J. Rothman, S. Greenland, and T. L. Lash, *Modern epidemiology*. Lippincott Williams & Wilkins, 2008.

[6] C. Mann, "Observational research methods. research design ii: cohort, cross sectional, and case-control studies," *Emergency Medicine Journal*, vol. 20, no. 1, pp. 54–60, 2003.

[7] P. R. Burton, D. G. Clayton, L. R. Cardon, N. Craddock, P. Deloukas, A. Duncanson, D. P. Kwiatkowski, M. I. McCarthy, W. H. Ouwehand, N. J. Samani, *et al.*, "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls," *Nature*, vol. 447, no. 7145, pp. 661–678, 2007.

[8] B. Goudey, D. Rawlinson, Q. Wang, F. Shi, H. Ferra, R. M. Campbell, L. Stern, M. T. Inouye, C. S. Ong, and A. Kowalczyk, "Gwis-model-free, fast and exhaustive search for epistatic interactions in case-control gwas," *BMC genomics*, vol. 14, no. Suppl 3, p. S10, 2013.

[9] G. Fang, M. Haznadar, W. Wang, H. Yu, M. Steinbach, T. R. Church, W. S. Oetting, B. Van Ness, and V. Kumar, "High-order snp combinations associated with complex diseases: efficient discovery, statistical power and functional interactions," *PloS one*, vol. 7, no. 4, p. e33531, 2012.

[10] S. Leem, H.-h. Jeong, J. Lee, K. Wee, and K.-A. Sohn, "Fast detection of high-order epistatic interactions in genome-wide association studies using information theoretic measure," *Computational Biology and Chemistry*, 2014.

[11] D. Sadava, D. Hillis, C. Heller, G. Orians, and W. Purves, *Life The Science of Biology*. Sinauer Associates, 8th ed., 2008.

[12] D. Uvehag, "Design and implementation of a computational platform and a parallelized interaction analysis for large scale genomics data in multiple sclerosis," 2013.

[13] D. Uvhage and H. Zazzi, "Geisa." `https://github.com/menzzana/geisa`, 2014.

[14] A. Ahlbom and L. Alfredsson, "Interaction: a word with two meanings creates confusion," *European journal of epidemiology*, vol. 20, no. 7, pp. 563–564, 2005.

[15] K. J. Rothman, *Epidemiology: an introduction*. Oxford University Press, 2002.

[16] A. Sjölander, W. Lee, H. Källberg, and Y. Pawitan, "Bounds on causal interactions for binary outcomes," *Biometrics*, 2014.

[17] A. Agresti, *Categorical data analysis*. John Wiley & Sons, Second ed., 2002.

[18] B. Lindgren, *Statistical theory*. CRC Press, fourth ed., 1993.

[19] H. T. O. Davies, I. K. Crombie, and M. Tavakoli, "When can odds ratios mislead?," *Bmj*, vol. 316, no. 7136, pp. 989–991, 1998.

[20] M. J. Knol, T. J. VanderWeele, R. H. Groenwold, O. H. Klungel, M. M. Rovers, and D. E. Grobbee, "Estimating measures of interaction on an additive scale for preventive exposures," *European journal of epidemiology*, vol. 26, no. 6, pp. 433–438, 2011.

[21] B. D. H. Foundation, "Smoking and oral health." `https://www.dentalhealth.org/tell-me-about/topic/sundry/smoking-and-oral-health`.

[22] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[23] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[24] Intel, "Intel unleashes its first 8-core desktop processor." `http://newsroom.intel.com/community/intel_newsroom/blog/2014/08/29/intel-unleashes-its-first-8-core-desktop-processor`, 2014.

[25] S. Almeida, "An introduction to high performance computing," *International Journal of Modern Physics A*, vol. 28, no. 22n23, p. 1340021, 2013.

[26] C. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[27] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective: Dynamics and Sustainability*. Springer, 2003.

[28] U. Drepper, "What every programmer should know about memory," 2007.

[29] F. Abi-Chahla, "Intel core i7 (nehalem): Architecture by amd?." `http://www.tomshardware.com/reviews/Intel-i7-nehalem-cpu,2041-2.html`, 2008.

[30] Intel, "i7-5960x specification sheet." `http://ark.intel.com/products/82930`, 2014.

[31] NVIDIA, *NVIDIA CUDA C Programming Guide*, v5.5 ed., July 2013.

[32] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[33] NVIDIA, "Nvidia tesla-kepler product description." `http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf`.

[34] NVIDIA, *NVIDIA CUDA C Best Practices Guide*, v5.5 ed., July 2013.

[35] Intel, "Intel Xeon Phi Coprocessor DataSheet, Document ID 328209 003EN." `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html`, 2014.

[36] M. H. NVIDIA, "Maxwell: The Most Advanced CUDA GPU Ever Made." `http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`, 2014.

[37] J. Poznanovic, "Plinkgpu: A framework for gpu acceleration of whole genome data analysis," Master's thesis, School of Informatics, University of Edinburgh, 2010.

[38] NVIDIA, "Kepler Tuning Guide." `http://docs.nvidia.com/cuda/kepler-tuning-guide`.

[39] Mark Harris, "How to Overlap Data Transfers in CUDA C/C++." `http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`.

[40] G. Ruetsch and B. Leback, "CUDA Fortran Asynchronous Data Transfers." `http://www.pgroup.com/lit/articles/insider/v3n1a4.htm`.

[41] NVIDIA, *CUBLAS Library User Guide*, v5.5 ed., July 2013.

[42] "CULA Tools: GPU Accelerated Linear Algebra," 2010.

[43] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, IEEE, 2010.

[44] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. Version 1.7.0.

[45] V. Volkov, "Unrolling parallel loops," *Tutorial at the*, p. 133, 2011.

[46] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[47] R. C. Martin, "Design principles and design patterns." `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, 2000.

[48] G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren, "Measuring reproducibility in computer systems research." `http://reproducibility.cs.arizona.edu/v1/tr.pdf`, 2013.

[49] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. "O'Reilly Media, Inc.", 2012.

[50] X.-H. Sun and Y. Chen, "Reevaluating amdahlâĂŹs law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183–188, 2010.

[51] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[52] NVIDIA, "Nsight eclipse edition." `https://developer.nvidia.com/nsight-eclipse-edition`.

[53] A. Gyenesei, J. Moody, A. Laiho, C. A. Semple, C. S. Haley, and W.-H. Wei, "Biforce toolbox: powerful high-throughput computational analysis of gene–gene interactions in genome-wide association studies," *Nucleic acids research*, vol. 40, no. W1, pp. W628–W632, 2012.

[54] L. S. Yung, C. Yang, X. Wan, and W. Yu, "Gboost: a gpu-based tool for detecting gene–gene interactions in genome–wide case control studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.

[55] Z. Zhu, X. Tong, Z. Zhu, M. Liang, W. Cui, K. Su, M. D. Li, and J. Zhu, "Development of gmdr-gpu for gene-gene interaction analysis and its application to wtccc gwas data for type 2 diabetes," *PloS one*, vol. 8, no. 4, p. e61943, 2013.

[56] S. Lee, M.-S. Kwon, I.-S. Huh, and T. Park, "Cuda-lr: Cuda-accelerated logistic regression analysis tool for gene-gene interaction for genome-wide association study," in *Bioinformatics and Biomedicine Workshops (BIBMW), 2011 IEEE International Conference on*, pp. 691–695, IEEE, 2011.

[57] S. Chikkagoudar, K. Wang, and M. Li, "Genie: a software package for gene-gene interaction analysis in genetic association studies using multiple gpu or cpu cores," *BMC research notes*, vol. 4, no. 1, p. 158, 2011.

[58] D. Sluga, T. Curk, B. Zupan, and U. Lotric, "Heterogeneous computing architecture for fast detection of snp-snp interactions," *BMC bioinformatics*, vol. 15, no. 1, p. 216, 2014.

[59] R. Jiang, F. Zeng, W. Zhang, X. Wu, and Z. Yu, "Accelerating genome-wide association studies using cuda compatible graphics processing units," pp. 70–76, 2009.

[60] M. D. Ritchie, L. W. Hahn, N. Roodi, L. R. Bailey, W. D. Dupont, F. F. Parl, and J. H. Moore, "Multifactor-dimensionality reduction reveals high-order interactions among estrogen-metabolism genes in sporadic breast cancer," *The American Journal of Human Genetics*, vol. 69, no. 1, pp. 138–147, 2001.

[61] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. Tang, and W. Yu, "Boost: A fast approach to detecting gene-gene interactions in genome-wide case-control studies," *The American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.

[62] Y. Zhang and J. S. Liu, "Bayesian inference of epistatic interactions in case-control studies," *Nature genetics*, vol. 39, no. 9, pp. 1167–1173, 2007.

[63] A. Albert, *Regression and the Moore-Penrose pseudoinverse*. Elsevier, 1972.

[64] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numerische Mathematik*, vol. 14, no. 5, pp. 403–420, 1970.

[65] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[66] D. F. Schwarz, I. R. König, and A. Ziegler, "On safari to random jungle: a fast implementation of random forests for high-dimensional data," *Bioinformatics*, vol. 26, no. 14, pp. 1752–1758, 2010.

[67] S. J. Winham, C. L. Colby, R. R. Freimuth, X. Wang, M. de Andrade, M. Huebner, and J. M. Biernacka, "Snp interaction detection with random forests in high-dimensional genetic data," *BMC bioinformatics*, vol. 13, no. 1, p. 164, 2012.

[68] S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*. Addison-Wesley, fifth ed., 2012.

[69] John, "How do i set the cuda stream cula uses?." `http://www.culatools.com/forums/viewtopic.php?f=14&t=1019`.

[70] "Magma documentation, function magmablassetkernelstream." `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/group__magma__util.html#ga0bbe77c58765b97bf03f589c43cb7ac1`.

[71] "Plink data format." `http://pngu.mgh.harvard.edu/~purcell/plink/data.shtml`.

Appendices

Appendix A

Lists

List of Algorithms

List of Examples

List of Figures

List of Tables

Appendix B

File Formats

B.1 PLINK Data Format

The PLINK files can be in different formats [71]. The format described here is the binary Plink files. It consists of three files. Bed, bim and fam [71]. The fam file contains the information about the individuals, the outcome, the persons and familys id, gender and parents id. The bim file contains the information about the SNPs but not their data. It has the SNPs id, chromosome, position on the chromosome and the alleles. The bed file has the actual SNP data in a binary format. Each pair of 0 and 1 is the genotype for one individual. See [71] for the exact details and information about the other Plink file formats.

B.2 Environmental and Covariates File Format

The environmental factors and covariates are stored in separate files with the same format, one file for environmental factors and one for the covariates if any. One of the columns needs to contain the individual ids, the rest of the columns should be data. The delimiter can be any reasonable string(e.g. something that is not part of a name or number) and there is an option to set the delimiter in the program. The default is tab delimited.

B.3 CuEira Result File Format

The results from CuEira are written in an csv file. The columns are described below.

snp_id
: The id of the SNP from the bim file

pos
: The row which the SNP had in the plink files

skip
: Numbers explaining why the SNP was excluded from calculation. 1=missing data. 2=low MAF, 3=low cell count, 4=negative position in bim file

risk_allele
The risk allele, not changed based on recoding

minor
The minor allele

major
The major allele

env_id
The id of the environmental factor, it is the column from the environmental file

no_alleles_X
The numbers of the alleles in group X

freq_alleles_X
The frequencies of the alleles group X

no_snpX_envY
Cell distribution with exposure X and Y

numberOfIterations
The number of iterations the LR model took to converge, one column for the multiplicative model and one for the additive

ap
The AP value for the additive model described in 2.1.4

reri
The RERI value for the additive model described in 2.1.4

OR
The odds ratios for the SNP, environmental factor and interaction. Add is for the additive model and mult is for the multiplicative. L is the lower confidence interval and H is the upper. The interval is from using the delta method [12]

recode
Case of recoding, 0=no recoding, 1=snp protective, 2=environment protective, 3=interaction protective

Each line after the header is a combination of a SNP and an environmental factor. Because of the parallel nature of the program the rows are not in any specific order. The column pos contains the row number that the SNP had in the plink files. This can be used to sort the data in.

The data in some columns are changed depending on the recoding. The multiplicative model is calculated using the recode from the additive model. The cell frequencies are also changed. However the risk allele is not changed.

Appendix C

Risk Allele Definition Differences

This appendix contains a more detailed look at different risk allele definitions than section 1.3.1.
The code for how JEIRA and GEISA calculates the risk allele is not the same as its definition [3, 12] talked about previously in section 1.3.1. The algorithm JEIRA and GEISA uses is shown in algorithm 10.

Algorithm 10: JEIRA and GEISA pseudo code to determine the risk allele

Data: $caseMaxAllele$ is the most common allele in case and
$controlMaxAllele$ is the most common in control
$caseMaxRatio$ and $controlMaxRatio$ is the frequency of
$caseMaxAllele$ and $controlMaxAllele$ respectively.

if $caseMaxRatio > controlMaxRatio$ **and**
$caseMaxAllele = controlMaxAllele$ **then**
  $riskAllele \longleftarrow caseMaxAllele$
else
  $riskAllele \longleftarrow caseMinAllele$

The allele frequencies can be shown in an 2×2 table. The columns are the frequencies for the different alleles and the rows are the groups(case and control). It can be split into four cases shown in table C.1. The four cases comes from the four cases of possible directions of the inequality between the maximum frequencies of the two groups, shown as the bold arrows. There is up, down, down diagonally and up diagonally, call them case 1 to 4. The rest of the inequalities can be found by using symmetry and that the sum of each row is 1.

By using the previously mentioned definitions in section 1.3.1 and the JEIRAs and GEISAs algorithm 10 what the risk allele is according to them for each case is shown in table C.2. Each definition correspond to examining the inequalities in the cases. For the definition using MAF the direction of the inequality

	Allele1		Allele2		Allele1		Allele2
Case	Max	→	Min		Max	→	Min
	↓	↘	↑		↑	↗	↓
Control	Max	→	Min		Max	→	Min
Case	Max	→	Min		Max	→	Min
	↓	↘	↑		↓	↖	↑
Control	Min	←	Max		Min	←	Max

Table C.1: *The four cases of allele frequencies. Case 1 upper left, case 2 upper right, case 3 lower left, case 4 lower right*

from case minimum to control minimum determines it. If it points towards control the the case minor allele is the risk allele. For JEIRA and GEISA if the maximum for case points straight down to the maximum for control then the major case allele is the risk allele. This only happens in case 1. For increased frequency in case it is the allele which has the inequality pointing straight down.

Definition	Case 1	Case 2	Case 3	Case 4
MAF based	Allele1	Allele2	Allele1	Allele2
JEIRA GEISA algorithm	Allele1	Allele2	Allele2	Allele2
Increase frequency in case	Allele1	Allele2	Allele1	Allele1

Table C.2: *Risk alleles for the cases in table C.1 based on the different definitions*

The risk allele is different for case 3 and case 4. For case 3 it is clear that it should be Allele1 because it has the highest frequency in case and also higher in case than control An extreme case of case 3 is shown in table C.3.

By examining a similar extreme case of case 4 it is clear that Allele1 should be the risk allele in that case too. The case is shown in table C.4. Almost everyone in the case group has Allele1 while no one in the control group does, it is unlikely that Allele2 would increase the risk in that case.

	Allele1	Allele2
Case	1	0.00
Control	0.01	0.99

Table C.3: *An extreme case of frequencies of case 3 in table C.1*

	Allele1	Allele2
Case	0.99	0.01
Control	0	1

Table C.4: *An extreme case of frequencies of case 4 in table C.1*

Appendix D

Bugs and Issues

This is a small collection of various bugs and limitations encountered in the course of the project that can be good to know for anyone that wants to compile CuEira or if the libraries are used in other projects.

D.1 Bugs

There are several problems with Intel compilers and libraries. Most libraries work fine with gnu C/C++ compiler but have problems with Intel compilers. Boost 1.55 does not work with Intel compilers, 1.54 works. AllOf in Google Test also does not work together with Intel compilers.

JEIRA prints the wrong results for the multiplicative model and as mentioned in appendix C the risk allele algorithm JEIRA uses is not the same as the definition.

D.2 Issues

Nvcc(i.e. the compiler for CUDA) can not compile C++11 code and no interface included in the code that nvcc compiles can use c++11. This problem can largely be avoided by using separate compilation. The CUDA find package module for CMake(included in version 2.8 and later) has a default flag that should be removed as it can cause problems. The flag tells nvcc to propagate the host code flags, if the compiler for the host code then uses C++11 that flag is propagated to nvcc and it will not compile The flag is disabled in CMake by using set(CUDA_PROPAGATE_HOST_FLAGS off).

Appendix E

How to Compile and Use CuEira

The source code for the program is available at github, `https://github.com/Berjiz/CuEira`.

List of dependencies. Listed version is the version used.

- Boost, tested with version 1.54, 1.55 does not work due to a bug with Intel compilers
- CMake, version 2.8
- CUDA, version 5.5
- MKL, version 13.1
- Intel compiler version 14.0
- Google Test and Google Mock, already included in the folder.

First install the dependencies, they might need to be compiled from source. Open a terminal and make a new directory where you want the program to be compiled. Enter the directory and write:
cmake /path/to/CuEira/
Then type:
make

The program should now be compiling. CMake migh not finding all dependencies. If it does you need to tell CMake where it is. The path to boost is set by using EXPORT BOOST_ROOT = /path/to/boost/ before using CMake. The path to the compilers is done with two options to the CMake command, CXX = /path/to/source for the C++ compiler and C = /path/to/source for the C compiler.

In the build/bin directory there is two executable files, CuEira and CuEira_Test. CuEira is the program itself while CuEira_Test runs all the tests for the program. The option -h or –help lists possible options for the program. The most

basic command to run CuEira is -b data -e environment file -x name of column with snp ids